Entwicklung eines Modellmanagementsystems als Bestandteil einer verteilten Simulationsarchitektur eines Forschungssimulators

Dem Fachbereich Maschinenbau
der Technischen Universität Darmstadt
zur
Erlangung des Grades eines Doktor–Ingenieurs (Dr.–Ing.)
eingereichte

D i s s e r t a t i o n

vorgelegt von

Dipl.–Ing. Carsten Huth
aus Rüsselsheim/M

Berichterstatter :	Prof. Dr.–Ing. W. Kubbat
Mitberichterstatter :	Prof. Dr.–Ing. R. Anderl
Tag der Einreichung :	28.05.2001
Tag der mündlichen Prüfung :	12.07.2001

D 17

Die vorliegende Arbeit entstand an dem von Herrn Prof. Dr.–Ing. W. Kubbat geleiteten Fachgebiet für Flugmechanik und Regelungstechnik der Technischen Universität Darmstadt im Zuge der Neuentwicklung einer Simulationsumgebung als Integrationsplattform für die vielfältigen Forschungsprojekte am Institut.

Ich danke Herrn Professor Dr.–Ing. W. Kubbat für die Unterstützung und Ermunterung, die er mir während der Erstellung dieser Arbeit zuteil werden ließ.

Für die freundliche Übernahme des Korreferates möchte ich mich bei Herrn Professor Dr.–Ing. R. Anderl bedanken.

Mein Dank gilt weiterhin den wissenschaftlichen Mitarbeitern des Fachgebietes und allen Studenten, die zum Gelingen dieser Arbeit beigetragen haben. Insbesondere Oliver Albert, Kai Engels und Jens Schiefele haben aufgrund ihrer Unterstützung und Begleitung all die Jahre hindurch wesentlichen Anteil an dieser Arbeit.

Nicht unerwähnt sollen auch meine „jüngeren“ Kollegen bleiben, die mir am Ende meiner Zeit am Fachgebiet den Rücken frei hielten.

Zuletzt möchte ich meinen Eltern danken, die mir mein Studium erst ermöglicht haben.

Ich versichere an Eides Statt, daß ich diese Arbeit mit Ausnahme der ausdrücklich erwähnten Hilfen selbständig durchgeführt habe.

Darmstadt, im Mai 2001

(Carsten Huth)

„Ich kann, weil ich will, was ich muß“

Immanuel Kant

Übersicht

Verteilte Simulation bedeutet die Kooperation einer Anzahl von Simulationsprozessen, die in einem Netzwerk von Rechnern ausgeführt werden. Jeder Prozeß erfüllt eine kleine, überschaubare Aufgabe, wobei die Summe der Aufgaben, das gemeinsame Ziel, über ein Simulationsmodell bestimmt ist. Ziel dieser Arbeit ist die Entwicklung einer Verwaltung von Simulationsprozessen, so daß diese im Zuge von Forschungsvorhaben in einer beliebigen Anzahl von Simulationsmodellen eingesetzt werden können.

Ausgehend von einer Betrachtung des Zyklus in der Entwicklung und Erstellung von Simulationsmodellen werden drei Hauptaufgaben definiert, in denen der Mensch (Benutzer, Entwickler und Administrator) entlastet und/oder unterstützt werden muß: die Integration der voneinander unabhängig entwickelten Simulationsprogramme in die Simulationsumgebung, die beliebige Kombination von Prozessen zu einer maßgeschneiderten Simulation sowie die Überwachung der erst zur Laufzeit feststehenden verteilten Prozeßstruktur.

Ansatzpunkt für die Umsetzung ist eine Analyse der Simulationsprogramme und -prozesse sowie eine Recherche bestehender Konzepte und Implementationen. Daraus folgend wird ein Komponentenmodell erarbeitet, welches die Eigenschaften, das Verhalten und die Beziehungen der Prozesse untereinander wiedergibt. Gleichzeitig werden Laufzeitaspekte der Prozesse mit in die Modellierung aufgenommen, um so jedem Prozeß zur Laufzeit die Rechnerumgebung zu gewährleisten, die von ihm zur Erfüllung seiner Aufgaben benötigt wird.

Für den Benutzer der Simulation wird ein auf dem Komponentenmodell basierendes Rahmenwerk, *Nemo's Model Organizer*, zur Verfügung gestellt, so daß dieser in die Lage versetzt wird, die komplexe Aufgabe einer Simulation zu bewältigen, ohne tiefergehende Kenntnisse zu besitzen. Dazu wird ein zentraler Arbeitsplatz in Form einer graphischen Bedienoberfläche bereitgestellt, mit dessen Hilfe Simulationsvorhaben konfiguriert und überwacht werden können.

Gleichzeitig wird das Modellmanagement an eine Datenbank angebunden, die als „Gedächtnis“ der Simulation betrachtet werden kann. Das in ihr gespeicherte Wissen ermöglicht einerseits die Benutzung der Simulation durch Nicht-Experten und kann andererseits auf diese Weise für nachfolgende Mitarbeiter am Institut erhalten bleiben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

ACE	Adaptive Communication Environment
ALSP	Aggregate Level Simulation Protocol
AM	Alias Manager
ANSI	American National Standard Institute
ASCII	American Standard Code for Information Interchange
ATM	Asynchronous Transfer Mode
CASE	Computer Aided Software Engineering
CCM	CORBA Component Model
CIDL	Component Implementation Definition Language
CIF	Component Implementation Framework
COM+	Component Object Model
CORBA	Common Object Request Broker Architecture
CORC	CORBA Components
CPU	Central Processing Unit
CRT	Cathod Ray Tube
DCOM	Distributed Component Object Model
DIS	Distributed Interactive Simulation
DLL	Dynamic Link Library
DMSO	Defense Modeling And Simulation Office
DoD	Department Of Defense (USA)
DSPA	Distributed Simulation Programming Architecture
ECAM	Electronic Centralized Aircraft Monitoring
EJB	Enterprise Java Beans
FCU	Flight Control Unit
FED	Federation Execution Data
FMRT	Fachgebiet Flugmechanik und Regelungstechnik
FOM	Federation Object Model

GUI	Graphical User Interface
HLA	High Level Architecture
HMI	Human Machine Interface
ICD	Interface Control Document
IDE	Integrated Development Environment
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronics Engineers
IIOP	Internet-Inter-ORB-Protocol
I/O	Input/Output
LCD	Liquid Crystal Display
ND	Navigation Display
NeCD	Nemo's Communication Database
NeMD	Nemo's Management Database
NeMO	Nemo's Model Organizer
NePD	Nemo's Program Database
NeSD	Nemo's Simulation Database
PC	Personal Computer
OBB	Octopus Box Builder
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker
OSIf	Octopus System Interface
OTW	Objekttechnologie-Werkbank
PFD	Primary Flight Display
QoS	Quality of Service
RAM	Random Access Memory
RMI	Remote Method Invocation
ROM	Read-Only Memory

RTI	Runtime Infrastructure
SGI	Silicon Graphics
SO	Shared Object
SOM	Simulation Object Model
SSE	Simulation Support Environment
STL	Standard Template Library
TAO	The ACE ORB
TCAS	Traffic Collision Avoidance System
UML	Unified Modeling Language
VR	Virtual Reality

Begriffserläuterungen

Administrator
Die Person, die das Rechnernetzwerk und die beiden Softwarepakete *Octopus* und *NeMO* verwaltet.

Benutzer
Die Person, die den Forschungssimulator zur Durchführung eines Versuches konfiguriert und bedient.

Cockpithardware
Die gesamte Hardware, die die Nachbildung des Flugzeugcockpits als Arbeitsplatz von Piloten ausmacht.

***DSPA*, Simulationsarchitektur**
Die Struktur der Simulationsumgebung am Fachgebiet Flugmechanik und Regelungstechnik, die von den beiden Softwarepaketen *Octopus* und *NeMO* umgesetzt wird.

Entwickler, Programmierer
Die Person, die Module entwirft und entwickelt, und dadurch Simulationsmodelle definiert.

Forschungscockpit
Cockpithardware und der Forschungssimulator vereinigt.

(Simulations-)Modul, Komponente
Definition einer klar abgetrennten Aufgabe als Teil eines Simulationsmodells.

***NeMO*, Modellmanagement, Modellverwaltung**
Die Software, die die Simulationsprogramme in die Simulationsarchitektur integriert und die dazugehörigen Prozesse zur Laufzeit verwaltet.

***Octopus*, Datenpool, Datenmanagement**
Die Software, die die Kommunikationsverbindung der Prozesse innerhalb der Umgebung herstellt.

(Simulations-)Programm
Umsetzung einer Teilaufgabe eines Simulationsmodells in Form eines Computer-Programms. Es können auch durchaus mehrere Teilaufgaben in einem Programm integriert sein.

Prozeß
Ein Prozeß ist die Instanz eines Programms. Ein Prozeß kann mehrfach ausgeführt werden, so daß mehrere Instanzen eines Programms existieren können.

Simulationsinfrastruktur
Das Rechnernetzwerk und die Software *Octopus*.

Simulationsmodell
Die Abbildung eines realen Systems unter Zuhilfenahme von Vereinfachungen und Verallgemeinerungen, die sich aus der gegebenen Fragestellung und den vorhandenen Möglichkeiten ergeben.

Das Simulationsmodell wird mit Hilfe einer Programmiersprache in eine Form überführt, die für eine Ausführung auf einem Rechner geeignet ist. Im allgemeinen wird ein Simulationsmodell nicht durch ein einzelnes Simulationsprogramm, sondern durch eine Menge von Simulationsprogrammen repräsentiert.

Simulationsprozeß
Instanz eines Simulationsprogramms und damit Teil des Simulationsmodells.

Simulationssoftware
Das Softwarepaket bestehend aus *Octopus* und den Simulationsprogrammen.

Simulator, Forschungssimulator
Die Simulationsarchitektur *DSPA*, die Simulationsprogramme und das Rechnernetzwerk.

Systemprozeß
Ein Systemprozeß ist ein Prozeß, der in irgendeiner Weise für den Betrieb der Simulationsarchitektur oder des Rechnernetzwerkes benötigt wird. Er steht in keinerlei Zusammenhang mit der inhaltlichen Simulationsaufgabe.

1 Einleitung

Hintergrund dieser Dissertation ist die Simulationsarchitektur *DSPA* (*Distributed Simulation Programming Architecture*) des Fachgebiets Flugmechanik und Regelungstechnik an der Technischen Universität Darmstadt, welche eine Struktur zur Entwicklung und zum Betrieb von verteilten Computer-Simulationen vorgibt. Verteilte Simulationen zeichnen sich dadurch aus, daß sie einerseits immer realitätsgetreuer und leistungsfähiger, andererseits aber auch immer komplexer und größer werden. In der Konsequenz sind sie für den „normalen“ Menschen kaum noch beherrschbar, welcher aber auf die Simulation als Mittel zum Zweck in Forschungsvorhaben angewiesen ist. Eine Unterstützung und Entlastung des Menschen ist daher unumgänglich.

In dieser Arbeit wird ein Modellmanagement als Bestandteil des gesamten Rahmenwerks der Simulationsarchitektur vorgestellt, welches die Vorteile der verteilten Simulation (Leistungsfähigkeit, realitätsgetreue Abbildung, quasi unbegrenzte Möglichkeiten in der Umsetzung, etc.) erhält und dem Menschen die Verwaltung und Organisation der Vielzahl der Prozesse eines Simulationsvorhabens abnimmt. Der Benutzer der Simulation benötigt kein tiefergehendes Wissen über die Simulationsumgebung, sondern kann sich vollständig auf sein Forschungsprojekt und die damit verbundenen Untersuchungsziele beschränken.

Der erste Teil der Einleitung beschäftigt sich exemplarisch mit dem institutseigenen Forschungssimulator. Es wird gezeigt werden, welche Nachteile und Probleme sich aus den hohen Ansprüchen hinsichtlich der Leistungsfähigkeit eines solchen Simulators ergeben können und welche Komplexität das Gesamtsystem „Simulator“ aufweist.

Ausgehend davon wird im Abschnitt Zielsetzung beschrieben, welche Aufgaben das Modellmanagement dem Menschen bei der Entwicklung und dem Einsatz von Simulationsmodellen abnimmt und welche Rahmenbedingungen dabei zu beachten sind. Des besseren Verständnisses wegen sei in diesem Zusammenhang darauf hingewiesen, daß die Simulationsarchitektur *DSPA* und das dazugehörige Rahmenwerk nicht mit einem vollständigen Simulator verwechselt werden dürfen ([Bäu98]). Erst durch die Integration eines Simulationsmodells in die Architektur erhält man einen funktionsfähigen Simulator ([NO95]). Die Art und Weise wie eine solche Integration zu erfolgen hat, wird in dieser Arbeit festgelegt.

Am Ende der Einleitung wird die Struktur der gesamten Arbeit im Überblick vorge-

stellt, anhand derer die gewählte Vorgehensweise zur Umsetzung der Aufgaben des Modellmanagements ersichtlich wird.

1.1 Motivation

Das Fachgebiet Flugmechanik und Regelungstechnik ist im Bereich der Luftfahrtforschung tätig. Ziel ist die Optimierung des gegenwärtigen und zukünftigen Luftverkehrssystems hinsichtlich Sicherheit, Leistungsfähigkeit und Wirtschaftlichkeit. Hauptansatzpunkt des Fachgebiets ist hierbei die Mensch-Maschine-Schnittstelle (Human-Machine-Interface (HMI)) *Cockpit*, die nach den heutigen Erfahrungen das größte Verbesserungspotential beinhaltet ([Kau98], [Cla99]).

Mögliche Fragestellungen sind unter anderem ([Ham97], [Kau98], [Pur00], [Hei99]):

- Wie läßt sich das Situationsbewußtsein des Piloten durch veränderte Gestaltung von Flugführungsanzeigen verbessern?
- Wie bekommt der Pilot eine verbesserte Rückmeldung über sein Handeln und über das tatsächliche Verhalten des Flugzeugs?
- Wie können die heutigen Verfahren und Prozeduren im Luftverkehr effizienter gestaltet und besser zwischen Bord (Pilot) und Boden (Air Traffic Control) verteilt werden?

Zum Zwecke der Beantwortung der oben beschriebenen Fragestellungen wurde am Fachgebiet Flugmechanik und Regelungstechnik ein Forschungssimulator und ein Flugzeugcockpit als Arbeitsplatz des Piloten aufgebaut. Mit Hilfe dieser Einrichtung, in ihrer Summe als Forschungscockpit bezeichnet, werden aussagekräftige Bewertungen von Neuentwicklungen auf dem Gebiet der Mensch-Maschine-Schnittstelle erst möglich.

Im Zuge der Untersuchung neuartiger Flugführungsanzeigen und Bedienelemente für das Cockpit eines Flugzeugs, werden hohe Anforderungen an einen Forschungssimulator gestellt. Die Hard- (Cockpit, Rechnernetzwerk, ...) und Softwarekomponenten (Simulationsmodelle, Analysetools, ...) werden ständig erweitert, modifiziert, verbessert, um den Ansprüchen aus allen Forschungsvorhaben des Fachgebiets gerecht zu werden. Zusätzlich zu diesen Strukturänderungen muß ein Forschungssimulator in der Lage sein, in den unterschiedlichsten Konfigurationen betrieben zu werden, damit, je nach durchgeführter Versuchsreihe, ein optimaler Versuchsaufbau

zur Verfügung gestellt werden kann. Schließlich steht über allem noch der Anspruch nach einer zufriedenstellenden Leistungsfähigkeit des Simulators, gerade im Hinblick darauf, daß in fast allen Fällen der Mensch in die Simulation einbezogen wird. Das bedeutet zumeist, daß die Simulation das zugrundeliegende Szenario nahezu realitätsgetreu wiedergeben muß.

Erfahrungen und Recherchen zeigen, daß eine Umsetzung aller Anforderungen ein Optimimierungsproblem darstellt ([NO95], [Boy96]), da sich die Anforderungen gegenseitig beeinträchtigen können. Probleme ergeben sich auch häufig daraus, daß nicht alle Anforderungen in der Entwurfs- und Umsetzungsphase gleichermaßen berücksichtigt werden, sondern erst aus einer später entstandenen Notwendigkeit heraus ([NO95], [Smi99]). Beispiele für die angedeuteten Problemstellungen sind:

- Die nahezu realitätsgetreue Abbildung der Wirklichkeit in der Simulation setzt Eigenschaften bezüglich „Quality of Service“ (QoS) voraus, wie beispielsweise eine zeitlich garantierte Zuteilung von Rechnerressourcen zur Abarbeitung von Prozessen oder die Synchronisation der beteiligten Prozesse hinsichtlich ihrer Ausführungsreihenfolge.

  Meist werden diese Anforderungen dadurch zu erfüllen versucht, daß ein System starr konzipiert und implementiert wird. Das kann z.B. bedeuten, das es nur in einer einzigen Konfiguration betrieben werden kann und darf, daß Prozesse eine dedizierte Hardware voraussetzen oder daß Kenntnisse über Ort und Verteilung anderer Prozesse fest in die Implementation eines Simulationsprozesses miteingearbeitet sind.

  In der Konsequenz sind Änderungswünsche, die in einem Forschungssimulator zwangsläufig auftreten, mit einem erheblichen Umbau- und/oder Rekonfigurationsaufwand verbunden.

- Flexible Systeme erfordern einen hohen Wissensstand der Benutzer, um die Leistungsfähigkeit des System vollständig ausnutzen zu können. Anders ausgedrückt bedeutet dies, daß die Benutzung kompliziert und aufwendig wird. Der Benutzer benötigt meist eine umfangreiche Anleitung, die in der Art eines „Kochrezeptes“ die Bedienung des Simulators vorgibt. Abweichungen von dieser Anleitung werden nicht gestattet, weil sonst die Funktionalität des Simulators in Frage gestellt ist.

  Ein weiterer Nachteil ergibt sich aus der Tatsache, daß an Universitäten die Mitarbeiter in der Regel nach fünf Jahren das Institut verlassen. Der Zyklus, in

dem neue Mitarbeiter in ein Arbeitsgebiet eingeführt werden müssen, ist also viel kürzer als in der Industrie oder anderen nicht-universitären Einrichtungen. Schafft man es nicht, in diesem Zyklus für eine Überlappung an Mitarbeitern zu sorgen, geht das angeeignete Wissen mit dem Weggang des Mitarbeiters verloren und muß mühsam wieder neu erarbeitet werden ([Kub99]).

- Während der Entwicklung von Simulatoren kann meist nicht die rasante technologische Entwicklung im Bereich Hardware vorausschauend in Betracht gezogen werden. Hinterher stellt man jedoch fest, daß man das gestiegene Leistungsvermögen neuer Rechner gerne nutzen möchte, um eine bessere Performance des Simulators zu erreichen, oder es wird von anderer Seite der Wunsch geäußert, eine neue Hardware, die für ein Forschungsvorhaben dringend erforderlich ist, in den bestehenden Simulator zu integrieren.

  Diese Flexibilität bezüglich der zugrundeliegenden Hardware des Simulators ist selten gegeben ([Dah98], [Boy96]), da Systeme häufig, z.B. aufgrund von Leistungsanforderungen, auf die verwendete Hardware abgestimmt werden. Im Extremfall ist der Simulator bei einem Wechsel der Hardware nicht mehr einsetzbar oder man bleibt für immer auf die Benutzung von alter Hardware fixiert.

- Fehlende Standardisierungen oder Richtlinien bezüglich der Struktur der Simulationsumgebung bedingen, daß bei Änderungen derselben häufig die gesamte Umgebung geändert oder Bestehendes nochmal implementiert werden muß.

  Ersteres ist der Fall, wenn beispielsweise die Änderung des Datenflusses eines Programms in der Simulation dazu führt, daß ebenso andere Programme geändert werden müssen, die Daten für dieses Programm zur Verfügung stellen oder von diesem Daten zur Verfügung gestellt bekommen (Schnittstellenproblematik).

  Letzteres ist der Fall, wenn keine saubere Trennung zwischen den Aufgaben der einzelnen Prozesse erfolgt (Modularisierung). Teilaufgaben werden unter Umständen derart miteinander verwoben, daß eine Überarbeitung der einen auch unweigerlich eine Adaption der anderen nach sich zieht, obwohl diese inhaltlich nicht miteinander in Beziehung stehen müssen.

Tabelle 1.1 faßt die Probleme und Wechselwirkungen noch einmal zusammen.

Gewünschter Effekt	Seiteneffekt
• Realitätsgetreue Abbildung („Virtual Environment“) • Hohe Leistungsfähigkeit mit „Quality of Service“	• Statisch vorbestimmte Prozeßstruktur und -verteilung • Erheblicher Umbau- und Rekonfigurationsaufwand
• Großer Leistungsumfang • Sehr detaillgetreu • Keine Einschränkung hinsichtlich Einsatz von Hard- und Software	• Hohe Komplexität des Gesamtsystems • Hohe Ansprüche an den Bediener • Unflexible Nutzung
• Nutzung spezifischer Hardwareeigenschaften z.B. im Hinblick auf Leistungsfähigkeit • Einsatz beliebiger, auch nicht Standard-Hardware	• Fixierung auf momentane Hardware • Mit der technologischen Weiterentwicklung kann nicht Schritt gehalten werden
• Simulation ohne Vorkenntnisse von allen nutzbar	• Wissen auf einige, wenige Experten verteilt • Wissen geht mit dem Weggang der Experten verloren
• Freiheit in der Umsetzung und Implementation	• Wiederverwendbarkeit bestehender Software schwierig • Häufiges Re-Engineering aufgrund fehlender Standards und Richtlinien nötig • Geringe Lebensdauer der erstellten Software

Tab. 1.1: Probleme und Wechselwirkungen

1.2 Zielsetzung

Vor dem Hintergrund der in Abschnitt 1.1 beschriebenen Problemstellungen ist das Ziel der vorliegenden Arbeit die Entwicklung eines Managements von Simulationsmodellen, welches die Nutzung der Vorteile einer modularen, verteilten Simulation weiterhin uneingeschränkt ermöglicht, aber gleichzeitig die unerwünschten Nebenwirkungen beseitigt oder minimiert. Ist dies nicht möglich, so sollen die Seiteneffekte durch das Modellmanagement zumindest abgefangen werden.

Die zentrale Aufgabe hierbei ist die Reduktion der an den Menschen gestellten Anforderungen, so daß dieser auch ohne Spezialwissen dazu in der Lage ist, die Simulationsumgebung als Hilfsmittel für seine Forschungsvorhaben zu verwenden. Der Vorgang der Entwicklung und des Einsatzes von Simulationsmodellen soll sich daher für den Menschen folgendermaßen darstellen.

Verteilte Computer-Simulation ist gleichbedeutend mit der Kooperation einer Anzahl von Simulationsprozessen, die in einem Netzwerk von Rechnern ausgeführt werden. Jeder Prozeß erfüllt eine kleine, überschaubare Aufgabe, wobei die Summe der Aufgaben, das gemeinsame Ziel, über das Simulationsmodell bestimmt ist. Ein Entwickler bekommt eine Teilaufgabe übertragen, auf die er sich voll und ganz konzentrieren soll; er benötigt keine Kenntnisse über andere Komponenten, sondern arbeitet vollständig entkoppelt von diesen. Im Zuge von Simulationsvorhaben können die getrennt entstandenen Module trotzdem beliebig miteinander kombiniert werden. Kooperation findet nur über einen standardisierten Kommunikationsmechanismus statt. Der Nutzer der Simulationsumgebung bekommt die Möglichkeit, ein für sein Vorhaben maßgeschneidertes Simulationsmodell dynamisch zu konfigurieren und ausführen zu lassen.

Die Verwendung der Komponenten setzt dabei kein Wissen über die Implementation oder die benötigte Laufzeitumgebung (z.B. Hardware-Anforderungen) voraus. Dadurch wird gewährleistet, daß auch andere außer dem Entwickler selbst, die Komponente benutzen können, und zwar so, wie es vom Entwickler vorgesehen worden ist. Gerade im Hinblick auf die Vererbung von Wissen an nachfolgende wissenschaftliche Mitarbeiter ist dies unbedingt erforderlich.

Aufgrund der Tatsache, daß vom Menschen nur geringe Kenntnisse über die Interna der Simulation erwartet werden dürfen, muß die Simulationsarchitektur ebenfalls Sorge dafür tragen, daß der Zustand der Simulationsprozesse, der Rechner und auch der Architektur selbst in ausreichendem Maße überwacht wird.

Der Ansatzpunkt zur Umsetzung der gerade vorgestellten Zielsetzungen besteht in der Betrachtung des Lebenszyklus eines Simulationsmodells. Anhand einer Analyse der Phasen, die zur Entwicklung und Erstellung von Simulationsmodellen durchlaufen werden, sollen diejenigen Vorgänge und Abläufe identifiziert werden, die unabhängig vom Modellinhalt immer wieder durchlaufen werden müssen. Anstatt daß ein Entwickler sie jedesmal von neuem abarbeitet, sind sie prädestiniert dafür, von einer grundlegenden Architektur bereitgestellt zu werden.

Die Fülle der Aufgaben wird auf die zwei Bestandteile der verteilten Simulationsarchitektur *DSPA*, das Daten- und das Modellmanagement, verteilt. Für das Modellmanagement ergeben sich hierbei drei große Aufgabenbereiche:

1. Integration

   Wie oben beschrieben, soll eine Komponente weitestgehend aus einer lokalen Betrachtungsweise heraus entstehen; ein Entwickler kann sich ganz auf seine „persönliche“ Problemstellung konzentrieren. In einem Schritt der Integration in die Simulationsarchitektur muß das neu entstandene Simulationsmodul überprüft und dahingehend vorbereitet werden, daß es später mit anderen Komponenten kombiniert und ausgeführt werden kann.

   Demzufolge stellen die im Integrationsvorgang gewonnenen Informationen eine Grundlage für die anderen beiden Aufgaben des Modellmanagements dar.

2. Konfiguration

   Die Simulationsarchitektur *DSPA* ist nicht auf ein Simulationsmodell festgelegt, sie kann prinzipiell beliebig viele Modelle verwalten. Aus diesem Grund muß das Modellmanagement eine Schnittstelle zur dynamischen Konfiguration des Forschungssimulators vorsehen, durch die der Benutzer in die Lage versetzt wird, festzulegen, welches Szenario simuliert werden soll.

   Als Ergebnis erhält man eine im Rechnernetzwerk verteilte Prozeßstruktur, die das gewünschte Simulationsmodell umsetzen soll.

3. Überwachung

   Die Praxis der Verifikation und Validation von Simulationsmodellen durch den Menschen in einem iterativen Prozeß vor der eigentlichen Laufzeit der Simulation, an derem Ende eine „Garantie“ für die Funktionsfähigkeit einer festen Simulatorkonfiguration steht, läßt sich in dieser Art nicht mehr anwenden. Der Grund hierfür ist die oben erwähnte dynamische Konfiguration. Dadurch steht

erst kurz vor der Laufzeit der Simulation fest, welche Softwaremodule zu einem Simulationsmodell zusammengesetzt werden sollen, und müssen demnach während der Ausführung überwacht und überprüft werden.

In der Erfüllung dieser Aufgaben muß ein äußerer Rahmen beachtet werden, der sich durch die Festlegung der Simulationsarchitektur auf bestimmte Grundprinzipien ergibt. Die Grundregeln der Simulationsarchitektur kann man im wesentlichen folgendermaßen charakterisieren:

- **modular**

  Ein monolithisches System ist den Anforderungen auf Dauer nicht gewachsen. Deswegen wird aus Gründen der Effizienz und Wiederverwendbarkeit ein modularer Forschungssimulator angestrebt, in dem die Gesamtaufgabe sich aus der Summe von vielen kleinen, überschaubaren Teilaufgaben zusammensetzt.

- **verteilt**

  Die Architektur muß in einem verteilten, heterogenem System von Rechnern betrieben werden können, da nur so gewährleistet ist, daß die zugrundeliegenden Hardware-Ressourcen einigermaßen problemlos den Bedürfnissen der Simulation entsprechend erweitert und modifiziert werden können.

- **flexibel**

  Ein Forschungssimulator hat keine festgelegte Standardsimulation, sein „Repertoire“ wird vielmehr durch die Vielfalt der Forschungsvorhaben bestimmt, in denen der Simulator zu Untersuchungszwecken eingesetzt wird. Diese Flexibilität muß von der Basis des Simulators, der Simulationsarchitektur, unbedingt gewährleistet werden.

- **skalierbar**

  Der volle Leistungsumfang des Forschungssimulators wird nicht immer für jede Untersuchung benötigt. Aus diesem Grunde muß der Simulator im Hinblick auf Schonung von bzw. im Hinblick auf zur Verfügung stehende Hardware-Ressourcen skaliert werden können, d.h. es muß möglich sein, nur Teile von Simulationsmodellen oder vereinfachte Versionen zu betreiben.

In den Ausführungen zur Motivation (vergleiche Kapitel 1.1) ist angedeutet worden, daß häufig der mit der Simulationsumgebung interagierende Mensch aufgrund der

Leistungsfähigkeit und Vielseitigkeit derselben schnell überfordert wird. Um dies zu vermeiden, muß das Modellmanagement auch eine Antwort auf die Fragestellung finden, wie der Mensch in der Interaktion mit der Simulation unterstützt werden kann. Dabei muß berücksichtigt werden, daß die Zielgruppe der Simulationsarchitektur nicht homogen hinsichtlich ihres Kenntnisstandes ist.

Vielmehr kann das Personenklientel in drei eigenständige Kategorien unterteilt werden, deren Sichtweisen und Vorstellungen jede für sich vom Modellmanagement unterstützt werden müssen:

- Der **Nutzer** ist diejenige Person, die die Simulationsumgebung benötigt, um Versuchsreihen zur Unterstützung verschiedenster Forschungsvorhaben durchzuführen. Er ist ein reiner Anwender, der eine möglichst einfache und intuitive Bedienung des Forschungssimulators bevorzugt, und in der Regel über wenig bis gar kein Hintergrundwissen verfügt.

- Die Gruppe der **Entwickler** umfaßt die Mitarbeiter in den Forschungsvorhaben, die durch ihre Forschungsziele festlegen, welche Szenarien der Forschungssimulator wiederzugeben in der Lage sein muß. Handelt es sich hierbei um neuartige oder erweiterte Szenarien im Hinblick auf das, was der Simulator zu leisten imstande ist, dann tragen die Entwickler mit von ihnen programmierten Simulationsmodulen zur Erweiterung der Simulationsmodelle bei.

- Der **Administrator** ist derjenige, der die Simulationsarchitektur betreut und wartet. Er ist dafür zuständig, den Forschungssimulator an Änderungen der Infrastruktur (z.B. neue Rechnerhardware, ...) anzupassen, ihn für die Benutzung durch andere vorzubereiten und ihn in Zusammenarbeit mit den Programmierern aus den unterschiedlichen Forschungsprojekten weiterzuentwickeln. Er nimmt damit eine Mittlerstellung zwischen der Gruppe der Benutzer und der Gruppe der Entwickler ein.

Abschließend sei darauf hingewiesen, daß über allen Anforderungen immer die Wahrung der Allgemeingültigkeit der Simulationsarchitektur und ihrer Bestandteile stehen muß. Architektonische Belange und Inhalte einer spezifischen Simulationsaufgabe dürfen **niemals** miteinander vermischt werden.

1.3 Struktur der Arbeit

Abbildung 1.1 zeigt die Struktur der vorliegenden Arbeit. Im Anschluß an dieses Kapitel wird der Kontext des Modellmanagements betrachtet und näher auf die Begriffe Simulationsmodell und Simulationsarchitektur eingegangen.

Darauf aufbauend werden die Anforderungen an ein Modellmanagement untersucht (Kapitel 3) und in einer Aufgabenbeschreibung zusammengefaßt. Mit diesem Wissen wird in Abschnitt 4 eine Analyse von Simulationsmodellen als das zu verwaltende „Objekt“ durchgeführt. Ziel ist zu erarbeiten, was die Umsetzung von Simulationsmodellen auf Computern mit Hilfe von Programmen überhaupt bedeutet. Womit hat man es zu tun, was sind die Bestandteile, was gilt es zu beachten, etc..

In den Kapiteln 5.1 bis 5.4 werden aktuelle Konzepte und Ansätze vorgestellt, die sich mit der gleichen Problematik auseinandersetzen, sowie einige beispielhafte Anwendungen, wo die Theorie in die Praxis umgesetzt worden ist. Die Recherche wird abgeschlossen durch eine zusammenfassende Diskussion der Vor- und Nachteile der beschriebenen Lösungsvorschläge (Abschnitt 5.5). Daraus wird eine Aussage abgeleitet, wie im vorliegenden Falle weiter zu verfahren ist.

Die Präsentation eines Modellmanagements für die verteilte Simulationsarchitektur *DSPA* als Ergebnis dieser Arbeit erfolgt in den Kapiteln 6 und 7. Dazu werden zuerst als Grundlage das entwickelte Rechnermodell und der eingeführte Signaturbegriff erläutert. Das zentrale Element des Modellmanagements stellt die Definition eines Komponentenmodells für Simulationsprogramme in der *DSPA* dar. Das Rahmenwerk zur Verwaltung der Simulationsprogramme und dessen Schnittstellen runden die Beschreibung des Modellmanagements ab.

Anschließend wird die gefundene Lösung einer Bewertung unterzogen (Kapitel 8), inwieweit sie der geforderten Aufgabenstellung entspricht, um dann in Kapitel 9 mit einer Zusammenfassung der Arbeit und einem Ausblick zu enden.

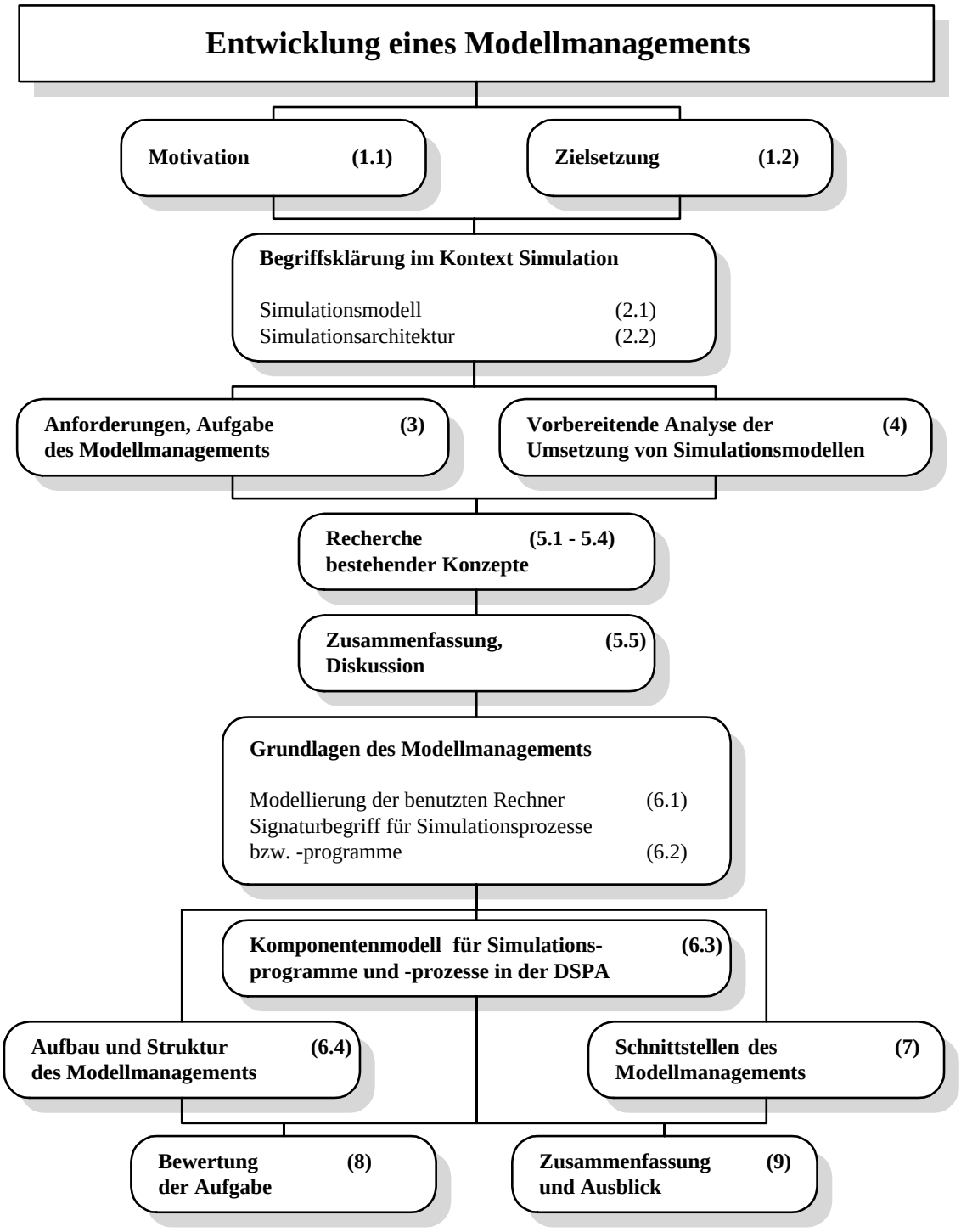


Abb. 1.1: Struktur der Arbeit

2 Simulationsmodell und -architektur

Das Institut Flugmechanik und Regelungstechnik führt Forschungsvorhaben zur Untersuchung und Verbesserung der Mensch-Maschine-Schnittstelle (HMI) im Cockpit heutiger und zukünftiger Verkehrsflugzeuge durch. Beispiele entsprechender Projekte finden sich unter anderem in [Alb01], [Eng01], [Kau98], [Pur00].

Unabdingbare Voraussetzung hierfür ist das Vorhandensein einer Simulationsumgebung, die die Gegebenheiten des Flugzeugcockpits den Ansprüchen entsprechend wiedergibt, so daß eine neue Mensch-Maschine-Schnittstelle aussagekräftig und kompetent beurteilt werden kann. An die Simulationsumgebung werden, wie schon in Abschnitt 1.1 beschrieben, sehr unterschiedliche und auch wechselnde Anforderungen gestellt. Die Konzeptionierung und Entwicklung eines Modellmanagements als Bestandteil der Simulationsumgebung am Fachgebiet Flugmechanik und Regelungstechnik soll Gegenstand der vorliegenden Arbeit sein. Der Kontext des Modellmanagements wird durch die beiden Begriffe *Simulationsmodell* und *-architektur* beschrieben.

Ersteres bezeichnet die Abbildung eines realen Systems unter Zuhilfenahme von Vereinfachungen und Verallgemeinerungen, die sich aus der gegebenen Fragestellung und den vorhandenen Möglichkeiten ergeben. Das Simulationsmodell wird mit Hilfe einer Programmiersprache in eine Form überführt, die für die Ausführung auf einem Rechner geeignet ist. Im allgemeinen wird ein Simulationsmodell nicht durch ein einzelnes Simulationsprogramm, sondern durch eine Menge von Simulationsprogrammen repräsentiert.

Die Bedeutung des Simulationsmodells liegt darin, daß es bzw. die dazugehörigen Prozesse die zu verwaltenden Einheiten darstellen. Deswegen wird in diesem Kapitel der Vorgang der Entwicklung und des Einsatzes von Simulationsmodellen vorgestellt, welcher sich grob in vier Arbeitsschritte unterteilen läßt: Konzeptionierung und Modellierung, Implementation und Überprüfung, Ausführung sowie Analyse.

Unter dem Begriff einer Architektur ist auf der anderen Seite eine Beschreibung zu verstehen, aus welchen Komponenten ein System aufgebaut ist und wie die Komponenten miteinander agieren können. Auf das Umfeld der Simulation angewendet, wird durch die Simulationsarchitektur festgelegt, wie eine Menge von Simulationsprozessen zur Durchführung eines Simulationsvorhabens kooperieren. Die Umsetzung der Vorgaben gehört zu den später genauer beschriebenen Aufgaben des Modellmanagements, welches ein Bestandteil des zur Architektur gehörigen Rahmen-

werks ist.

Die Betrachtung endet mit der Vorstellung der verteilten Simulationsarchitektur *DSPA* (*Distributed Simulation Programming Architecture*) des Fachgebiets Flugmechanik und Regelungstechnik. Dazu gehört eine Beschreibung ihrer Randbedingungen und ihres Leistungsumfanges sowie die Aufteilung in die zwei großen Bereiche *Daten-* und *Modellmanagement*. Ersteres ist Gegenstand von [Eng01], letzteres soll in dieser Arbeit entwickelt werden.

2.1 Simulationsmodell

2.1.1 Definition und Begrifflichkeit

Simulation kann ganz allgemein als ein Prozeß definiert werden, in dessen Verlauf ein Modell der Realität oder eines imaginären Systems erstellt wird, um dann mit diesem Modell Untersuchungen bzw. Experimente durchführen zu können ([Smi99], [And00]).

Die Gründe für die Verwendung einer Abbildung anstelle des realen Systems sind vielfältiger Natur:

- Direkte Untersuchungen am realen System sind nicht möglich oder zu teuer.
- Erst durch Modelle werden manche reale Zusammenhänge einer Analyse zugänglich.
- Durch ein Modell kann man sich auf die wesentlichen, d.h. die interessierenden Eigenschaften eines realen Systems beschränken.
- Unüberschaubare Systeme können durch Modellbildung so weit abstrahiert werden, daß der Mensch sie vollständig erfassen kann.

Die Erstellung eines Modells ist unweigerlich mit Vereinfachungen und Verallgemeinerungen verbunden ([And00]), da entweder die realen Sachverhalte nicht vollständig bekannt oder bestimmte Aspekte für die betrachtete Fragestellung von untergeordneter Bedeutung sind, und deswegen vernachlässigt werden können.

Nichtsdestoweniger ist ein Modell ein sehr nützliches Hilfsmittel, da es zuallererst eine einheitliche Beschreibung für die Betrachtung eines Systems schafft. Auf dieser Basis lassen sich dann Systeme verstehen, analysieren und vergleichen. Außerdem

wird der Betrachter in die Lage versetzt, Aussagen über das reale System zu machen ([Smi99], [And00]).

Abhängig von der Ausführung des Simulationsmodells kann man Simulation in zwei große Bereiche unterteilen ([NO95], [Sch85]):

- Diskrete Simulation.
- Kontinuierliche Simulation.

Ersteres ist der Fall, wenn alle Zustandsänderungen innerhalb der Simulation nur diskret, d.h. sprunghaft erfolgen können, letzteres, wenn alle Zustandsübergänge in der Simulation stetig verlaufen.

Diskrete Simulation ist die heutzutage vorherrschende Art der Simulation und meint fast immer die Ausführung der Simulation auf einem Digital-Rechner mit Hilfe eines Computer-Programms. Nach [BGK+97] wird dann das eigentliche Simulationsmodell als das konzeptionelle Modell und das Computer-Programm als das Implementationsmodell bezeichnet. Eine kontinuierliche Simulation hingegen wäre beispielsweise die Simulation eines Feder-Masse-Dämpfer-Systems mit Hilfe eines elektronischen Schwingkreises. Aufgrund einer Analogie in den Differentialgleichungen, die beide Systeme beschreiben, kann bei Kenntnis des einen Systems der Zustand des anderen für jeden Zeitpunkt vorhergesagt werden.

Im Falle der vorliegenden Arbeit wird das Simulationsmodell ebenfalls mit der Hilfe von Computer-Programmen umgesetzt, es handelt sich also auch hier um diskrete Simulation. Im folgenden wird eine kleine Beispielsimulation vorgestellt, damit sich der Leser einen besseren Eindruck über die Art der Simulation am Fachgebiet Flugmechanik und Regelungstechnik verschaffen kann.

2.1.2 Implementationsbeispiel

Aufgabe des Forschungssimulators ist es, dem Piloten seinen Arbeitsplatz, das Flugzeugcockpit, zu simulieren. Der dem Menschen gegenüber sichtbare Teil besteht in dem Nachbau der Hardware des Cockpits selbst (s. Abbildung 2.1 und [ARE+98]).

Im linken Bild sieht man eine Totalansicht des Cockpits, welches sich in seinen Maßen an denen eines Airbus A340 Flugzeugs orientiert. Das Cockpit ist modular aufgebaut, so daß die verschiedenen Bereiche wie Piloten- und Copiloten-Arbeitsplatz, Mittelkonsole, etc. relativ einfach ausgetauscht und durch andere Bauteile ersetzt

Abb. 2.1: Das Forschungscockpit des Fachgebiets FMRT

werden können. Das rechte Bild zeigt das Cockpit während eines Einsatzes aus der Sicht des Piloten.

Die Instrumentierung des Cockpits erlaubt dem Piloten sowohl manuelle als auch automatisierte Flugverfahren durchzuführen. Man erkennt folgende, wesentliche Bereiche:

- Die Flugführungsanzeigen Primary Flight Display (PFD), Navigation Display (ND) und Electronic Centralized Aircraft Monitoring (ECAM) Displays, welche auf den sechs LC-Displays im vorderen Teil des Cockpits dargestellt werden.

- Die primären Eingabelemente der Piloten, nämlich die zwei aktiven Sidesticks zur linken und zur rechten Seite des Cockpits, die Pedale unterhalb der Flugführungsanzeigen-Displays sowie die Mittelkonsole, wo sich z.B. die Schubhebel zur Steuerung der Motoren befinden.

- Sekundäre Eingabelemente wie die Flight Control Unit (FCU), über welche der Autopilot im automatischen Flug bedient werden kann.

- Der Overhead-Bereich des Cockpits, wo ein Großteil der Systembedienung des Flugzeugs stattfindet. Die eigentlich im Cockpit realer Flugzeuge verwendete Hardware (Panels, Schalter, etc.) wird hier durch ein touchsensitives LC-Display emuliert.

- Neuentwicklungen für den zivilen Luftverkehr, wie z.B. das aus dem militärischen Bereich bereits bekannte Head-Up-Display, welches dem Piloten

Flugzustandsinformation in die Cockpit-Scheibe einspiegelt, so daß der Pilot seine Aufmerksamkeit nicht von der Sicht nach außen abwenden und den Flugführungsanzeigen im Inneren zuwenden muß.

Auf diesen beiden Bildern ist das zugehörige Sichtsystem nicht sichtbar, welches für den Piloten die Sicht aus dem Cockpit heraus generiert. Es besteht aus drei Projektoren, einem Projektionsmedium sowie einem Hohlspiegel, über welchen die Piloten auf das Projektionsmedium blicken. Der Sichtbereich beträgt 180° horizontal und 20° vertikal. Es sei auch noch vermerkt, daß es sich um eine kollimierte Bilddarstellung handelt, die die realen Gegebenheiten wesentlich besser wiedergibt als normale Projektionseinrichtungen.

Die für die Simulation dieses Arbeitsplatzes benötigten Software-Module sind in Abbildung 2.2 dargestellt.

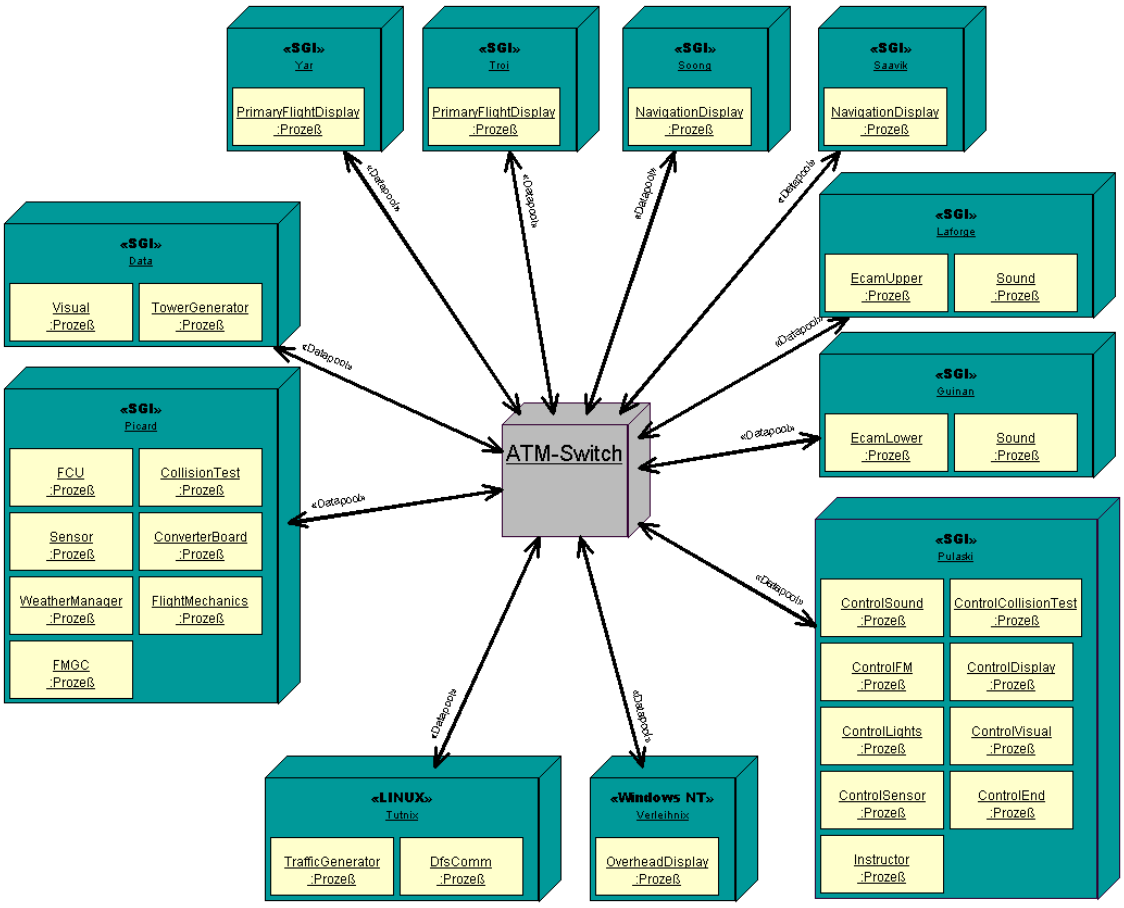


Abb. 2.2: Anzahl und Verteilung der Prozesse einer Beispielsimulation

In diesem Falle wird das Simulationsmodell mit der Hilfe von 29 eigenständigen Prozessen auf 11 verschiedenen Rechnern ausgeführt. Der Austausch von Informationen zwischen den Prozessen zur Laufzeit findet über eine Kommunikationseinrichtung namens *„Datenpool“* ([Eng01]) statt, welcher in Kapitel 2.2.2 überblicksartig beschrieben wird. Die auf diese Art und Weise in einem verteilten Rechnernetzwerk aufgebaute Prozeßstruktur stellt hinsichtlich ihrer Beherrschung nicht gerade triviale Anforderungen an den Betreiber der Simulation.

Bei den Rechnern handelt es sich hauptsächlich um Workstations der Firma Silicon Graphics (SGI), die über einen ATM-Switch miteinander verbunden sind. In diesem Beispiel wird jeweils nur ein LINUX-Rechner („Tutnix“), sowie ein Windows NT-Rechner („Verleihnix“) eingesetzt. In Zukunft dürfte der Einsatz dieser Plattformen jedoch deutlich erhöht werden.

2.1.3 Lebenszyklus

Entgegen den Anfängen in der Entwicklung und dem Betrieb von Simulationen, wo nur einige wenige Experten das entsprechende Wissen zum Aufbau einer Simulation besaßen, kann man heute einen fest umrissenen Prozeß definieren, wie man von einer anfänglichen Formulierung einer Problemstellung zu einer simulationstechnischen Umsetzung desselben kommt ([Smi99]).

Dieser Prozeß wird als *Simulations-Entwicklungs-Prozeß* ([Smi99]) oder auch als *Lebenszyklus des Simulationsmodells* ([NO95]) bezeichnet, wobei in dieser Arbeit der Begriff Lebenszyklus verwendet werden soll.

Der in Abbildung 2.3 bildlich dargestellte Lebenslauf eines Simulationsmodells läßt sich grob in vier Abschnitte untergliedern, welche iterativ durchschritten werden:

1. Konzeptionierung und Modellierung
2. Implementation und Überprüfung
3. Ausführung
4. Analyse

In der ersten Phase wird abhängig von einer Problemfelddefinition eine Aufgabe gestellt, für deren Lösung ein Modell eines realen Systems und ein Konzept zur softwaretechnischen Umsetzung des Modells erarbeitet werden muß. Dieses Konzept wird

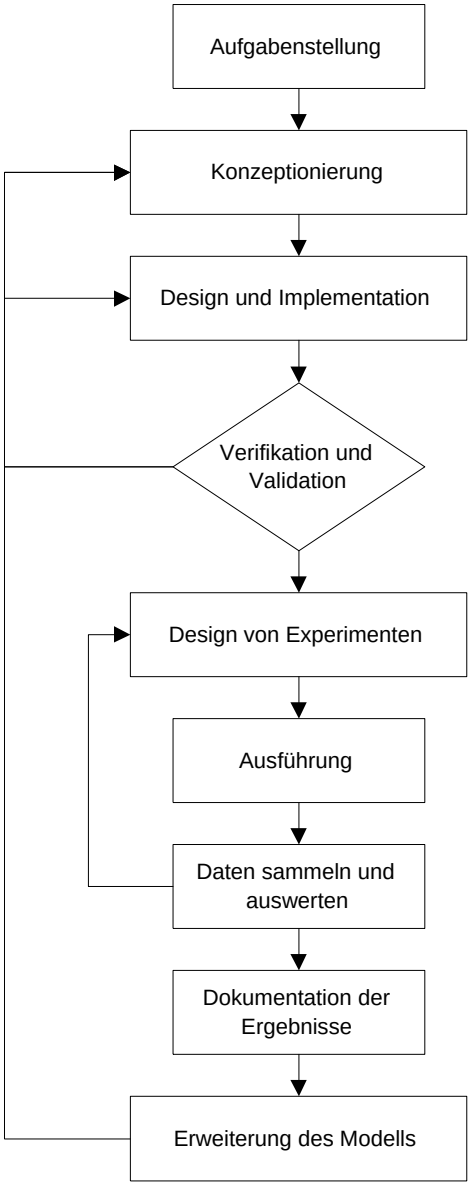


Abb. 2.3: Lebenszyklus eines Simulationsmodells

in der zweiten Phase in Form von Computer-Programmen implementiert, worauf im Anschluß eine Validation und Verifikation der erstellten Software erfolgt. In unserem Kontext bedeutet Validation die Überprüfung, ob das Simulationsmodell das zu untersuchende physikalische Modell realitätsgetreu wiedergibt. Verifikation hingegen meint die Überprüfung, ob die Umsetzung des Simulationsmodells in Software auch entsprechend der Konzeptionierung erfolgt ist ([Smi99]). Treten in der Überprüfungsphase Fehler oder Unstimmigkeiten auf, so kann das einen Rückschritt zur Konzeptionierung und Modellierung nach sich ziehen. Hieran läßt sich der iterative Charakter des Lebenszyklus erkennen. Wird jedoch als Ergebnis der Überprüfung die Implementation des Simulationsmodells im Rahmen der Aufgabenstellung akzeptiert, kann man im Anschluß (dritte Phase) damit beginnen, Experimente zu definieren und das Simulationsmodell unter den entsprechenden Randbedingungen auszuführen. Die letzte Phase beschäftigt sich mit der Analyse der in den Simulationsläufen gewonnenen Daten. Das kann aus zwei verschiedenen Perspektiven heraus geschehen. Die erste ist der eigentliche Zweck der Simulation, man möchte Aussagen über das untersuchte reale System treffen und bedient sich dabei des Hilfsmittels der Simulation. Die gewonnenen Daten beschreiben im Rahmen der Abbildungsgenauigkeit die Wirklichkeit. Die gesammelten Daten können jedoch auch eine Bewertung und Beurteilung des Simulationsmodells selbst nach sich ziehen, weil beispielsweise festgestellt wird, daß gewisse Aspekte in der Abbildung der Wirklichkeit fehlerhaft oder unberücksichtigt gelassen worden sind. Hier ergibt sich also ein weiterer Ansatzpunkt zur Iteration mit der Möglichkeit eines Rücksprungs in die Phasen Konzeptionierung und Modellierung bzw. Implementation und Überprüfung.

Für die Entwicklung der Simulationsumgebung am Fachgebiet Flugmechanik und Regelungstechnik soll der Lebenszyklus des Simulationsmodells als Orientierung und Leitfaden dienen. An ihm läßt sich erkennen, wie und wo der Mensch in der Entwicklung, dem Aufbau und dem Betrieb der Simulation am besten und effektivsten unterstützt werden kann. Mit der Zielvorgabe, den Menschen den Umgang mit der Simulation so weit wie möglich zu erleichtern, kann man die Simulationsumgebung am Fachgebiet Flugmechanik und Regelungstechnik auch als *Simulation Support Environment* (SSE) charakterisieren. Nach [NO95] subsummiert ein Simulation Support Environment eine integrierte Umgebung für die Entwicklung von Simulationsmodellen sowie ein computer-unterstütztes Simulationssystem, in welchem das Simulationsmodell ausgeführt werden kann, und deckt damit alle Phasen im Lebenslauf eines Simulationsmodells ab.

Der ideale Zustand, daß alle Phasen im oben beschriebenen Lebenszyklus durch eine

integrierte Entwicklungs- und Ausführungsumgebung unterstützt werden, ist jedoch gegenwärtig eher die Ausnahme als die Regel. Meistens gibt es entsprechende Umgebungen nur für einzelne Phasen des Modell-Lebenslaufes und von verschiedenen Herstellern, aber keine integrierte Gesamtlösung.

2.2 Simulationsarchitektur

Die Komplexität großer objekt-orientierter Softwaresysteme und die gleichzeitige Forderung nach gesteigerter Produktivität, kurzen Entwicklungszeiten und hoher Qualität der produzierten Software zwingt den Entwickler heutzutage, Strukturierungs- und Konstruktionsansätze zu verfolgen, die einen komplexen Anwendungsbereich so in Teilbereiche organisiert, daß diese getrennt voneinander entwickelt und realisiert werden können ([Bäu98], [GLL+99]). Auf diese Art und Weise lassen sich die Prinzipien der objekt-orientierten Vorgehensweise (Wiederverwendung, einfache und folgenlose Änderung von Teilbereichen, inkrementelle Software-Entwicklung) erst in die Tat umsetzen.

Für die Realisierung der Simulationsumgebung am Fachgebiet Flugmechanik und Regelungstechnik sind obengenannte Ansätze berücksichtigt worden. Bevor die Übertragung der beschriebenen Methodik in Form der verteilten Simulationsarchitektur *DSPA* (*Distributed Simulation Programming Architecture*) beschrieben werden kann, sollen noch die Begriffe *Architektur* und *Rahmenwerk* (*„Framework“*) eingeführt werden.

2.2.1 Architektur und Rahmenwerk

Architektur

Eine Architektur beschreibt auf abstrakte Art und Weise die Komponenten, aus denen ein System aufgebaut ist, die Interaktionen zwischen diesen Komponenten sowie Muster, wie das System aus den Komponenten aufgebaut werden kann ([Bäu98], [GLL+99]).

Sie bietet eine Orientierung und Anleitung bei der softwaretechnischen Realisierung von Anwendungen, indem sie den anfallenden Koordinations- und Abstimmungsaufwand im Aufbau und dem Ablauf einer Simulation zu bewältigen hilft. Die Basis dafür ist eine ganzheitliche Betrachtung des Systems mit dem Ziel, die Grundstruktur und den Kontrollfluß für die vollständige Anwendung einheitlich festzu-

legen ([And00], [GLL+99]), und dadurch die Kombinier- und Integrierbarkeit der entwickelten Komponenten zu garantieren.

Der Geltungsbereich von Architekturen beschränkt sich nicht unbedingt auf eine explizite Anwendung, sondern kann durchaus für die Umsetzung einer Reihe von ähnlichen Problemen eingesetzt werden. Diese Eigenschaft wird unter anderem mit dem Begriff *domänen-spezifisch* ([GLL+99], [Smi99]) bezeichnet. Die gewonnene Flexibilität läßt sich leicht nachvollziehen, wenn man bedenkt, daß mit einer Architektur so verschiedene Dinge wie Simulation von Prozeßketten in der Herstellung oder Verkehrsflußsimulationen realisiert werden können ([Smi99]).

Rahmenwerk

Nach [Bäu98] stellt ein Rahmenwerk eine softwaretechnische Lösung für eine Gruppe ähnlicher Probleme zur Verfügung. Ebenso wie eine Architektur legt ein Rahmenwerk das Zusammenspiel (den Kontrollfluß) von Komponenten hinsichtlich des zu lösenden Problems fest. Der Unterschied zur Architektur besteht darin, daß es sich bei Rahmenwerken um Implementationen und nicht um abstrakte Beschreibungen handelt.

Ein Rahmenwerk implementiert nur die Grundstruktur einer Anwendung, nicht die vollständige Anwendung selbst. Diese ergibt sich abhängig von dem betrachteten Problem durch eine Spezialisierung der Grundstruktur an den dafür vorgesehenen Stellen. Es läßt sich also festhalten, daß ein Rahmenwerk auch die Stellen, an denen es erweitert, spezialisiert und angepaßt werden kann, selbst definiert ([GLL+99]). Ein Beispiel für eine Komponente, die sinnvollerweise in einem Rahmenwerk implementiert werden sollte und häufig auch wird, ist die Benutzerschnittstelle von interaktiven Anwendungen, um auf diese Weise eine einheitliche Form der Präsentation und der Handhabung (engl. *„look and feel“*) dem Nutzer gegenüber zu garantieren ([GLL+99]).

Rahmenwerke können hierarchisch aufgebaut sein, d.h. Rahmenwerke können Komponenten übergeordneter Rahmenwerke sein.

2.2.2 Die verteilte Simulationsarchitektur *DSPA*

Die verteilte Simulationsarchitektur *DSPA* beschreibt den Aufbau der Simulationsumgebung am Institut Flugmechanik und Regelungstechnik. Dieser Aufbau ist aus der Analyse des in Kapitel 2.1.3 beschriebenen Lebenszyklus hervorgegangen. Die

Aufgaben, die zur Erstellung und Ausführung eines Simulationsmodells erfüllt werden müssen, werden in *modellspezifische und allgemeine Aufgaben* unterschieden. Modellspezifisch heißt, daß es eine Aufgabe ist, die von dem zu implementierenden Simulationsmodell abhängt. Allgemein drückt aus, daß es sich um eine Aufgabe handelt, die im Zusammenhang mit jedem Simulationsmodell erfüllt werden muß, also nicht direkt von der eigentlichen, simulierten Problemstellung betroffen wird.

Die allgemeinen Aufgaben in der Simulationsarchitektur *DSPA* sollen dann extrahiert und in einem Rahmenwerk implementiert werden, um so dem Menschen ein *Simulation Support Environment* (SSE) für die Flugsimulation am Fachgebiet Flugmechanik und Regelungstechnik zur Verfügung zu stellen.

Konzeptionelle Randbedingungen

Eine Architektur gibt nach obiger Definition eine Orientierung und Anleitung bei der softwaretechnischen Realisierung von Anwendungen und legt für den Programmierer einen äußeren Rahmen fest, innerhalb dessen dieser sich bewegen kann. Die Grundprinzipien der verteilten Simulationsarchitektur lassen sich im wesentlichen folgendermaßen charakterisieren:

- **modular**

  Die bereits vor der Entwicklung der *DSPA* gemachten Erfahrungen mit dem Einsatz eines Flugsimulators für Forschungsprojekte am Institut Flugmechanik und Regelungstechnik haben gezeigt, daß Simulationsmodelle parallel in den verschiedenen Projekten entwickelt werden müssen. Aus Gründen der Effizienz ist deswegen eine Modularisierung des Forschungssimulators unbedingt angeraten. Modularisierung heißt in diesem Zusammenhang, daß das Simulationsmodell in kleine, überschaubare Pakete aufgeteilt wird, welche getrennt voneinander umgesetzt werden können. Diese Pakete werden im folgenden mit dem Begriff *„Softwaremodul“* bezeichnet. Erst durch die Modularisierung wird die Wiederverwendung von Komponenten bereits bestehender Simulationsmodelle möglich und Änderungen von Komponenten wirken sich nur noch lokal auf das Simulationsmodul und nicht auf die komplette Anwendung aus. Außerdem ist die Wartung einzelner, überschaubarer Module deutlich einfacher als die großer monolithischer Strukturen. Es soll an dieser Stelle schon darauf hingewiesen werden, daß die Modularisierung einen erhöhten Koordinationsaufwand während der Entwicklung und dem Betrieb eines Simulationsmodells

nach sich zieht, was jedoch durch das dem Menschen zur Verfügung gestellte Rahmenwerk abgemildert werden soll.

- **verteilt**

  Ein jedes Simulationsmodell stellt individuelle Anforderungen hinsichtlich der benötigten Hardware-Ressourcen (Computer). Um diesen Anforderungen gerecht zu werden, geht die Simulationsarchitektur *DSPA* von einer Simulation in einem verteilten, heterogenen System aus (Rechner mit verschiedenen Betriebssystemen), weil nur auf diese Weise gewährleistet ist, daß die zur Verfügung stehende Hardware einigermaßen problemlos den Bedürfnissen der Simulation entsprechend erweitert und modifiziert werden kann. Damit wird dem Trend weg von Superrechnern (monolithische Struktur) hin zu Rechnernetzwerken (verteilte Struktur) Rechnung getragen.

  Ein verteiltes System versetzt den Nutzer außerdem in die Lage, die Rechner am Fachgebiet Flugmechanik und Regelungstechnik optimal auszulasten, was im Hinblick auf die Tatsache, daß das Rechnernetzwerk am Institut parallel für die wissenschaftlichen Arbeiten der Mitarbeiter und der Studenten eingesetzt wird, von nicht zu unterschätzender Bedeutung ist.

  Die verteilte Ausführung der Simulation muß für den oder die Entwickler eines Simulationsmodells vollkommen transparent sein, d.h. die Konzeptionierung und Implementation eines Simulationsmoduls darf nicht von einer bestimmten Verteilung der Module zur Zeit der Ausführung abhängig sein.

- **flexibel**

  Ein Forschungssimulator hat keine festgelegte Standardsimulation, die einmal entwickelt und dann nur noch benutzt wird. Er muß vielmehr ein „Repertoire“ von Simulationen zur Verfügung stellen, welches durch die Vielfalt der Forschungsvorhaben, in denen der Simulator eingesetzt werden soll, bestimmt wird. Davon stellt jedes einzelne Simulationsmodell in der Regel eine maßgeschneiderte Kombination von Simulationsmodulen dar, die in ihrer Gesamtheit das zu untersuchende reale System der Aufgabe entsprechend wiedergeben. Häufig erfordert ein maßgeschneidertes Modell auch die Änderung oder Neuentwicklung von Softwaremodulen, welche nach ihrer Überprüfung auf Richtigkeit ohne großen Zeit- und Arbeitsaufwand in die Simulationsumgebung integriert und betrieben werden sollen.

Das Rahmenwerk der *DSPA* muß diese Flexibilität unbedingt gewährleisten und der Entwicklung und der Ausführung von Simulationen so wenig wie möglich Restriktionen auferlegen. Da nicht alle Ansprüche und Eventualitäten im Voraus bedacht werden können, muß daß Rahmenwerk bei Bedarf weiterentwickelt und adaptiert werden können, sollte also ein offenes System darstellen.

- **skalierbar**

  Der volle Leistungsumfang des Forschungssimulators wird nicht immer für jede Untersuchung benötigt. Aus diesem Grunde muß der Simulator im Hinblick auf Soft- und Hardware skaliert werden können. Ersteres heißt, daß es möglich sein muß, nur Teile eines Simulationsmodells oder vereinfachte Versionen ausführen zu lassen, letzteres bedeutet, daß man eine Simulation sowohl mit einer kleinen Anzahl Hochleistungsrechner als auch mit einer großen Anzahl Standardrechner durchführen können muß.

  Der Skalierbarkeit werden in der Realität unter anderem dadurch Grenzen gesetzt, daß die Ausführung eines Softwareprogramms die spezielle Hardware eines bestimmten Rechners benötigt, z.B. werden für die Darstellung der Außenansicht für die Piloten im Flugzeugcockpit besonders leistungsfähige Graphik-Computer gebraucht, die zusätzlich noch an ein entsprechendes Projektionssystem angeschlossen sein müssen.

  Genaue Aussagen, ab welcher Anzahl von verwendeten Softwaremodulen und Rechnern die Skalierbarkeit nicht mehr gewährleistet werden kann, können bei dem gegenwärtigen Kenntnisstand am Fachgebiet Flugmechanik und Regelungstechnik nicht getroffen werden. Deswegen wird die Forderung nach Skalierbarkeit vorerst so weit abgemildert, daß zumindest das Konzept, welches hinter der Simulationsarchitektur *DSPA* steht, keine Begrenzung in der Anzahl von Softwaremodulen und Rechnern vorsehen darf.

Leistungsumfang

Der Leistungsumfang der verteilten Simulationsarchitektur *DSPA* orientiert sich daran, welche allgemeinen Aufgaben während der Entwicklung eines Simulationsmodells durch ein Rahmenwerk übernommen werden können. Abbildung 2.4 zeigt den für die Simulationsarchitektur *DSPA* adaptierten Lebenszyklus eines Simulationsmodells, wobei der gold- und der grünfarbige Teil den Verantwortungsbereich

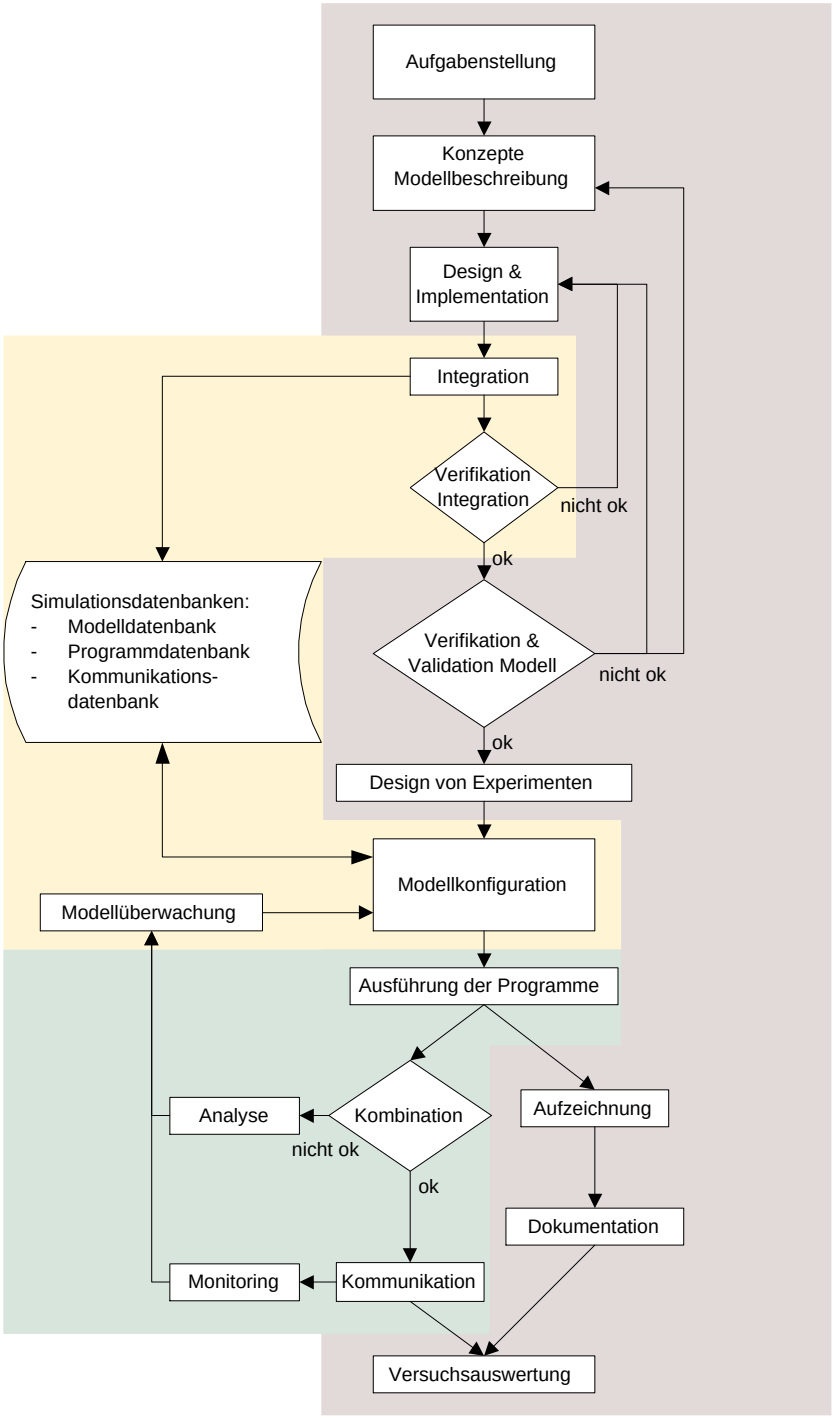


Abb. 2.4: Von der DSPA adaptierter Lebenszyklus des Simulationsmodells

des Rahmenwerkes der *DSPA* deutlich machen.

Ausgehend von der in Kapitel 2.1.3 vorgenommenen Unterteilung des Lebenszyklus lassen sich folgende Aufgaben für die *Distributed Simulation Programming Architecture* (*DSPA*) identifizieren:

1. Konzeptionierung und Modellierung

   Während dieser Phase sind hauptsächlich modellspezifische Belange betroffen. Die Simulationsarchitektur kann nur Richtlinien und Schnittstellen vorgeben, damit die Simulationsmodule später in die Architektur integriert und zu einem ausführbaren Gesamten kombiniert werden können. Diese Vorgaben betreffen folgende Bereiche:

   - Der Grad der Modularisierung, häufig auch als *„Granularität“* ([Bäu98]) bezeichnet, sollte so hoch sein, daß ein Simulationsmodul ohne große Probleme von einzelnen Programmierern überschaut werden kann.
   - Der Datenaustausch zwischen den Simulationsprozessen zur Laufzeit wird von der Simulationsarchitektur übernommen. Dem Entwickler wird dazu ein *„Leitbild“* ([Bäu98]) und eine zu verwendende Programmierschnittstelle an die Hand gegeben.
   - Die Gesamtheit aller in einer Simulation austauschbaren Daten wird zu Informations- und Dokumentationszwecken von einer zentralen Koordinationsstelle verwaltet, auf die alle Entwickler und Benutzer der Simulation jederzeit Zugriff haben. Der Zugriff ist nicht auf die Laufzeit eines Simulationsvorhabens beschränkt.
   - Benutzerschnittstellen für Simulationsprozesse sollen eine einheitliche Präsentation und Handhabung für die gesamte Simulation berücksichtigen. Außerdem soll die Bedienungslogik von der eigentlichen Simulationsaufgabe getrennt implementiert werden, so daß später die Bedienung der verteilten Simulation von einem zentralen Arbeitsplatz aus erfolgen kann.

2. Implementation und Überprüfung

   Die Implementation eines Modells ist ganz dem Entwickler überlassen. Die einzige Restriktion seitens der *DSPA* erfolgt aufgrund der zu verwendenden

Programmierschnittstelle für die Kommunikation zwischen den Simulationsprozessen zur Laufzeit (zum gegenwärtigen Zeitpunkt steht diese für die Programmiersprachen *C/C++* auf den Plattformen *SGI IRIX*, *LINUX* und *Microsoft Windows NT* zur Verfügung).

Da nur der Entwickler wissen kann, welches reale System mit welcher Abbildungsgenauigkeit durch die Implementation seines Moduls wiedergegeben werden soll, kann die Verifikation des einzelnen Moduls nicht von der Simulationsarchitektur geleistet werden. Sie tritt erst dann in Aktion, wenn es darum geht, das entsprechende Simulationsmodul in das Rahmenwerk der *DSPA* zu integrieren. Das Ziel der Integration ist eine Überprüfung aller Module und deren Organisation in einer zentralen Datenbank, auf die in den späteren Schritten Ausführung und Analyse zurückgegriffen werden soll.

Dazu müssen alle Informationen gesammelt werden, die für die Kombination des betrachteten Moduls mit anderen bestehenden zu einem Simulationsmodell und die Ausführung der dazugehörigen Simulationsprogramme von Bedeutung sind. Quelle dieser Information ist einerseits der Entwickler des Moduls selbst, andererseits werden sie in Testläufen gewonnen, was impliziert, daß die untersuchte Komponente mit anderen kombiniert und dann ausgeführt wird.

Die Integration ist dann erfolgreich, wenn alle benötigten Informationen gewonnen werden konnten und keine Konflikte mit anderen Komponenten aufgetreten sind. Von diesem Zeitpunkt an steht das betreffende Simulationsmodul für alle Simulationsvorhaben zur Verfügung. Die Definition, was ein Konflikt ist und wie er sich auswirken kann, wird im weiteren Verlauf der Arbeit behandelt werden.

Im Anschluß an die Integration können dann Simulationsmodelle für Experimente zusammengestellt werden. Die Tauglichkeit des Simulationsmodells bezüglich der untersuchten Fragestellung kann auch hier wieder nur vom Nutzer, nicht von der Architektur beurteilt werden. Die verteilte Simulationsarchitektur *DSPA* unterstützt den Menschen aber dahingehend, daß verifizierte und validierte Simulationsmodelle zum Zwecke der Wiederverwendbarkeit in einer Datenbank abgelegt werden können.

3. Ausführung

   Grundlage für die Durchführung einer Simulation ist eine entsprechende Vorbereitung seitens des Menschens, die die Ziele und Absichten des

Experiments festlegt. Ausgehend davon wird auf die oben beschriebene Simulationsprogramm-Datenbank zurückgegriffen und entschieden, welche Simulationsmodule zu einer Simulation zusammengesetzt werden sollen. Dazu gehört auch die Entscheidung, auf welchen Rechnern im Netzwerk die verschiedenen Programme gestartet werden sollen. Für die Erfüllung dieser komplexen Aufgabenstellung bekommt der Nutzer erstens Unterstützung und zweitens Einblick in die verteilte Simulationsarchitektur *DSPA*. Dieser Aufwand ist jedoch nicht immer nötig, da auf vorkonfigurierte Simulationen zurückgegriffen werden kann, wo dann die Kombination und Positionierung der einzelnen Simulationsmodule transparent für den Nutzer vorgenommen wird.

Die eigentliche Ausführung der Simulation besteht nun darin, alle Simulationsprogramme zu starten, die Kommunikationsstruktur für den Austausch der Informationen zwischen den einzelnen Simulationsprozessen aufzubauen und alle Programme quasi-parallel auf den Computern im Netzwerk des Fachgebiets Flugmechanik und Regelungstechnik abzuarbeiten. In den Verantwortungsbereich der *DSPA* fallen hierbei

- die Vorbereitung und Durchführung des Datenaustauschs sowie
- die Analyse und Überwachung der verteilten Simulationsumgebung (Prozesse, Kommunikationsstruktur und Rechnernetzwerk).

Mithilfe der durch die Analyse und das Monitoring der Simulationsumgebung gewonnenen Daten ist der Mensch in der Lage, die Güte der Simulation hinsichtlich des Versuchsvorhabens zu beurteilen. Ein Beispiel hierfür wäre die Information über die Größe der durch die Kommunikation verursachten Totzeiten, weil diese sich in einer interaktiven Simulation negativ auf den Probanden auswirken können. Zusätzlich sollen die bereitgestellten Informationen dazu dienen, im Fehlerfall die Lokalisierung und Bereinigung der Fehlerursache möglichst schnell und reibungslos zu gestalten.

Die Aufzeichnung und Dokumentation von Versuchsdaten liegt nicht in der Verantwortung der Simulationsarchitektur, weil diese keinerlei Wissen besitzt und auch nicht besitzen soll, welche Daten für die inhaltliche Analyse eines Experiments von Bedeutung sind. Aus der Sicht der *DSPA* sind alle Tools zur Aufzeichnung und Dokumentation nichts weiter als ganz normale Simulationsmodule.

4. Analyse

   Eine Analyse eines Simulationsmodells bzw. eines Experiments oder seiner selbst kann das Rahmenwerk der Simulationsarchitektur *DSPA* nicht eigenständig durchführen; dies kann nur der mit der Simulationsumgebung vertraute Benutzer. Der Vorgang kann jedoch unterstützt werden, indem die Simulationsarchitektur soviel Information über sich und die verwendeten Simulationsmodule sammelt, daß der Mensch einen genügend tiefen Einblick in das aufgebaute verteilte System bekommt, um die eigentliche Analyse ohne Probleme anstellen zu können.

Daten- und Modellmanagement

Aus der obigen Beschreibung des Leistungsumfanges der *DSPA* lassen sich zwei Organisationseinheiten herauskristallisieren (siehe Abb. 2.5).

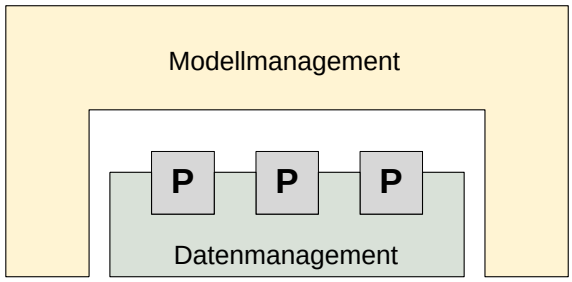


Abb. 2.5: Organisationseinheiten in der DSPA

1. Datenmanagement

   Das Datenmanagement sorgt für den Austausch von Informationen zwischen den Simulationsprozessen zur Laufzeit. Es läßt sich weiter unterteilen in den Aufbau und die Verwaltung einer verteilten Kommunikationsstruktur, die Überwachung des Datenaustauschs sowie den Austausch selbst.

   Entwickler von Simulationsmodulen bekommen eine Programmierschnittstelle geboten, mit der sie auf die Funktionalitäten des Datenmanagements zugreifen können, und einen konzeptionellen Rahmen, innerhalb dessen die Schnittstelle ihre Gültigkeit besitzt.

   Für eine detaillierte Betrachtung des Datenmanagements in der verteilten Simulationsarchitektur *DSPA* sei an dieser Stelle auf [Eng01] verwiesen.

2. Modellmanagement

   Das Modellmanagement verwaltet ein oder mehrere Simulationsmodelle im Hinblick auf ihre Benutzung in Simulationsvorhaben. Dazu gehört zuallererst die Integration der am Fachgebiet Flugmechanik und Regelungstechnik entwickelten Simulationsmodule und deren Organisation in einer zentralen Simulationsprogramm-Datenbank. Auf diese Datenbank soll dann zum Zwecke der Konfiguration eines gewünschten Simulationsmodells und dessen Ausführung in einem verteilten Rechnernetzwerk zugegriffen werden können. Die Unterstützung der Konfiguration und Ausführung sowie eine Überwachung der dadurch festgelegten Prozeßstruktur in der Simulation ist Aufgabe des Modellmanagements.

   Die Entwicklung eines solchen Modellmanagements im Rahmen der verteilten Simulationsarchitektur *DSPA* soll Gegenstand dieser Arbeit sein. Ausgangspunkt ist das folgende Kapitel, welches das Ziel hat, unter Berücksichtigung der Anforderungen und Randbedingungen am Fachgebiet Flugmechanik und Regelungstechnik eine detaillierte Aufgabenbeschreibung für ein Modellmanagement zu erarbeiten.

3 Modellmanagement

Im vorangegangenen Kapitel wurde erläutert, wie sich die verteilte Simulationsarchitektur *DSPA* in die zwei Bereiche Datenmanagement und Modellmanagement aufteilt und welche Verantwortungsbereiche den beiden Organisationseinheiten zugeordnet werden können. Für das Modellmanagement sollen die Aufgaben nun eingehender betrachtet werden.

Die unmittelbare Aufgabe besteht zuerst darin, entkoppelt voneinander entwickelte Simulationsprogramme in die Architektur zu integrieren und sie auf Kombinierbarkeit mit anderen Programmen zu überprüfen. Die so gewonnenen Informationen bilden die Grundlage, auf der eine beliebige Menge von Simulationsprozessen zur Laufzeit über das Modellmanagement ausgewählt und dynamisch zu einer Simulationssession zusammengestellt werden können. Die Überwachung der verteilten Prozeßstruktur und der Betriebsmittel wie beispielsweise der Rechner soll ebenfalls online vom Modellmanagement übernommen werden.

Die Erfüllung der Aufgaben des Modellmanagements erfolgt in Zusammenarbeit mit dem Menschen. Daraus ergibt sich die Notwendigkeit, diesem eine Schnittstelle zum Modellmanagement zu bieten. Ziel ist ein graphisch-interaktiver, zentraler Arbeitsplatz, von dem aus der Mensch in den Ablauf der Simulation und in die Arbeitsweise des Modellmanagements eingreifen kann.

In diesem Zusammenhang muß beachtet werden, daß das Klientel des Modellmanagements in Gruppen mit unterschiedlichen Kenntnissen (Nutzer, Entwickler und Administrator) aufgeteilt wird, so daß der Mensch eine seinen Fähigkeiten und seinem Wissen entsprechende Unterstützung erfahren kann.

Die zweite, nicht minder wichtige Schnittstelle des Modellmanagements ergibt sich aus der Forderung nach einer Anbindung an eine Datenbank. Das Wissen über die Simulation kann nicht immer dann gewonnen werden, wenn es benötigt wird, und muß deswegen dauerhaft und extern festgehalten werden.

Die Aufgabenbeschreibung wird abgerundet durch die Festlegung von Rahmenbedingungen und Richtlinien, die das Modellmanagement in Ergänzung zur *DSPA* vorgibt, um eine optimale Aufgabenerfüllung seinerseits zu ermöglichen. Dazu gehört unter anderem die Definition der Systemgrenzen für das Modellmanagement, die besagen, daß Simulationsprogramme bzw. -prozesse immer nur als *Black Box* betrachtet werden dürfen. Außerdem wird für jeden Simulationsprozeß die Trennung

der eigentlichen Simulationsaufgabe von Aufgaben zur Ablaufsteuerung des Prozesses vorgeschrieben.

3.1 Aufgabenbeschreibung

Für den Betrieb und die Benutzung von Computer-Simulationen gelten die gleichen Prinzipien wie für den Einsatz anderer großer Softwaresysteme, die jedoch in den theoretischen Betrachtungen von Simulationen häufig unbedacht bleiben. Beispiele hierfür sind ([Smi99]):

- Unzureichende Berücksichtigung einer benutzerfreundlichen Bedienung und Verwaltung der Simulation.
- Die Flexibilität hinsichtlich der Kombinationsmöglichkeiten von Komponenten zu einer Simulation geht meist nicht einher mit einem entsprechenden Konfigurationsmanagement. Größere Simulationsumgebungen werden dann recht schnell nicht mehr beherrschbar, insbesondere im Zuge von Weiterentwicklungen, wenn von den Simulationsmodulen verschiedene Versionen vorliegen.
- Anleitung und Unterstützung durch das System selbst ist nur rudimentär vorhanden, so daß nur die eigentlichen Entwickler mit der Simulationsumgebung umzugehen in der Lage sind.
- Architektonische Gesichtspunkte werden in der Konzeptionierung und dem Design wenig in Betracht gezogen, was redundante Entwicklungen und einen geringen Wiederverwendungsanteil von bereits bestehenden Komponenten nach sich ziehen kann.

Auch in [NO95] wird die Problematik einer integrierten Entwicklungsumgebung für Computer-Simulationen angesprochen. Danach zeichnet sich ein *„Simulation Support Environment“*, zu denen sich die Simulationsarchitektur *DSPA* zählt, durch das Vorhandensein von Tools wie unter anderem ein *„Premodel Manager“* und ein *„Project Manager“* aus. Ersteres hat die Aufgabe die Wiederverwendbarkeit bereits existierender Simulationsmodell-Komponenten zu gewährleisten, letzteres beinhaltet die relativ unspezifische Aussage, daß eine Simulation nichts anderes als ein Projekt ist, worauf die Erfahrungen und Methodiken aus dem Bereich des Projektmanagements weitestgehend anwendbar sind.

Vor diesem Hintergrund und der globalen Leitlinie, den Menschen im Umgang mit der Simulation zu unterstützen, sollen die Aufgaben des Modellmanagements in der verteilten Simulationsarchitektur *DSPA* festgelegt werden. Diese ergeben sich aus einer eingehenderen Analyse des in Kapitel 2.2.2 beschriebenen Leistungsumfanges der *DSPA*.

3.1.1 Integration

Die Implementation eines Simulationsmodells in der *DSPA* besteht aus einer Summe von Einzelkomponenten, den Simulationsprogrammen. Zur Ausführung hat man es dann nicht mit einem monolithischen Programm, sondern vielmehr mit einer Vielzahl von Programmen zu tun, die eine komplexe, verteilte Struktur im Rechnernetzwerk des Fachgebiets Flugmechanik und Regelungstechnik aufspannen. Die Vielzahl von Programmen wird von einer Mehrzahl von Entwicklern implementiert, die alle voneinander entkoppelt arbeiten. Entkopplung heißt, daß der Entwickler weitestgehend von der Notwendigkeit befreit ist, Kenntnisse über die anderen Simulationsprogramme eines Simulationsmodells besitzen zu müssen. Stattdessen arbeitet ein Programmierer nur mit der Vorstellung eines zentralen *„Datenpools“* ([Eng01]), der ihm als alleinige Schnittstelle zum Austausch von Daten dient. Den Datenpool kann man sich wie einen gemeinsamen Speicher vorstellen, auf den die Prozesse zur Laufzeit zum Zwecke des Lesens und Schreibens zugreifen. Datenaustausch über direkte Verbindungen zwischen den Prozessen in der Simulation findet nicht statt.

Aufgabe des Modellmanagements ist es nun, die getrennt voneinander entwickelten Simulationsmodule in das Rahmenwerk der *DSPA* zu integrieren. Als Ergebnis der Integration sind die Programme in einer zentralen Datenbank verfügbar und können in frei wählbaren Kombinationen zu einem maßgeschneiderten Simulationsszenario zusammengesetzt werden.

Voraussetzung für die Erfüllung der Aufgabe ist die Definition eines *Eincheckvorgangs*, der verbindlich festlegt, welche Schritte zur Integration eines Moduls in das Rahmenwerk durchlaufen und welche Informationen dabei festgehalten werden müssen.

Der Eincheckvorgang ist aus zwei Gründen von besonderer Bedeutung:

- Ein erfolgreiches Durchlaufen desselben bescheinigt einem Simulationsprogramm die Beachtung der Rahmenbedingungen der verteilten Simulations-

architektur *DSPA* und die Kombinierbarkeit mit anderen in der zentralen Programm-Datenbank vorhandenen Modulen.

- Er ist eine von zwei möglichen Informationsquellen, auf die sich das Modellmanagement zur Erfüllung seiner Aufgaben stützen kann.

  (Die zweite ist das Datenmanagement ([Eng01]), worauf erst später genauer eingegangen werden soll).

3.1.2 Konfiguration

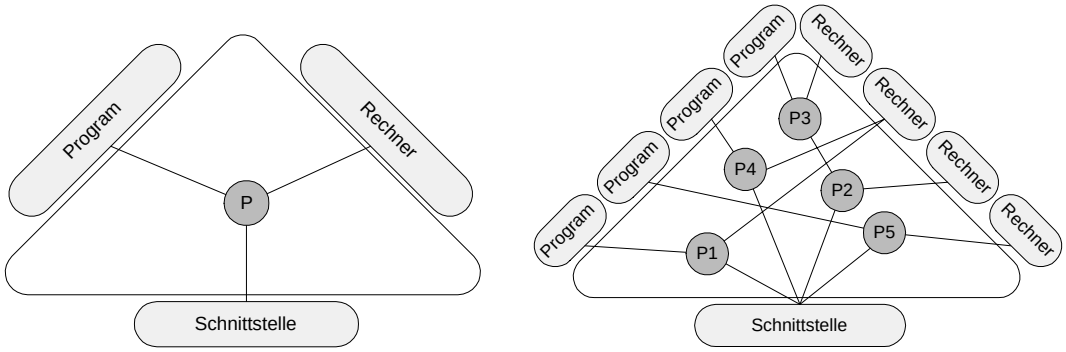


Abb. 3.1: Ideal- und Realsituation der Konfiguration einer Simulation

Die ideale Situation der Bedienung eines Simulators ist diejenige, daß man über **eine Schnittstelle einen Prozeß auf einem Rechner** ausführen muß (siehe Abbildung 3.1).

Diese Situation ist aufgrund der Tatsache, daß es sich um

- ein verteiltes Rechnernetzwerk und
- eine Vielzahl von voneinander abhängigen Prozessen handelt,

nicht mehr gegeben. Der Bediener muß im Extremfall aus der Vielzahl der zusätzlich noch versionierten Simulationsprogramme die richtige Auswahl für ein Simulationsvorhaben treffen und diese auf geeigneten Rechnern im Netzwerk des Fachgebiets zur Ausführung bringen. Zusätzlich können auch noch durch die Auswahl der Prozesse weitere Subprozesse betroffen sein, die ebenfalls zur Ausführung der Simulation benötigt werden.

Man kann jedoch versuchen, dem Benutzer zur Lösung der Aufgabe Unterstützung zukommen zu lassen, so daß es sich für ihn nahezu wie die Idealsituation verhält, den Menschen also von komplizierten Konfigurationsaufgaben weitestgehend entlasten.

Die Teilaufgaben, die dabei vom Modellmanagement übernommen werden sollen, beinhalten:

- Vorbereitung der Simulationsarchitektur, was im wesentlichen bedeutet, das Datenmanagement und die Rechner im Netzwerk auf Betriebsbereitschaft hin zu überprüfen.
- Kombinieren einzelner Simulationsmodule zum gewünschten Simulationsmodell bzw. Verwaltung bereits vorkonfigurierter Simulationsmodelle.
- Verteilung der durch ein Simulationsmodell festgelegten Prozesse im Rechnernetzwerk.
- Eigentliche Ausführung der Programme.
- Sammeln von Analysedaten, die für die Erfüllung der Aufgaben des Modellmanagements gebraucht werden (alle modellspezifischen Daten sind hiervon ausgeschlossen, da sie nicht in den Verantwortungsbereich der Simulationsarchitektur fallen).

3.1.3 Überwachung

Der grundsätzliche Unterschied zu herkömmlichen Simulatoren ist die Tatsache, daß erst kurz vor der Laufzeit feststeht, welche Simulationsprozesse zur Ausführung kommen sollen. Das Zusammenspiel der Prozesse kann also nicht mehr während der Entwicklung durch den Menschen verifiziert und validiert, sondern muß zur Laufzeit überprüft werden.

Da die Benutzung des Simulators aber nicht auf „Experten“ begrenzt werden soll, kann nicht vorausgesetzt werden, daß der Mensch in der Lage ist, die Überwachung zu übernehmen. Folglich muß das Modellmanagement diese Funktionalität zur Verfügung stellen.

In ansteigender Abstraktion lassen sich vier Ebenen der Überwachung definieren:

- Zustand der verwendeten Ressourcen (Rechnernetzwerk)
- Funktion eines einzelnen Prozesses

- Zusammenspiel aller Prozesse
- Summe aller der über das Datenmanagement ausgetauschten Daten

Ziel der Überwachung ist die Detektion und die Beseitigung oder, falls nicht möglich, die Anzeige von Fehlern. Außerdem soll der Benutzer, auch bei fehlerfreiem Betrieb, einen Einblick in den Zustand des Systems bekommen, was quasi als Nebenprodukt der Überwachung abfällt.

3.1.4 Benutzerschnittstelle

Das Selbstverständnis des Modellmanagements sieht vor, daß nur in Zusammenarbeit mit dem Menschen die Simulationsarchitektur *DSPA* ihren vollen Nutzen entfalten kann:

- Der Mensch macht Vorgaben zur Ausführung der Simulation.
- Der Mensch will in den Ablauf der Simulation eingreifen.
- Der Mensch wird zur Erfüllung der Aufgaben des Modellmanagements gebraucht, z.B. als Informationsquelle.
- Der Mensch möchte Informationen über und einen Einblick in die Simulation.

Erfahrungen mit dem vorherigen Flugsimulator am Fachgebiet Flugmechanik und Regelungstechnik haben gezeigt, daß eine dezentralisierte Bedienung und Administration desselben durch eine rein textliche Ein- und Ausgabe bei steigender Komplexität nur noch von Experten wahrgenommen werden kann. Zu den Aufgaben des Modellmanagements gehört deswegen die Bereitstellung eines zentralen Arbeitsplatzes, von dem aus der Mensch mit Hilfe einer graphischen Benutzeroberfläche (*Graphical User Interface*) mit dem Modellmanagement interagieren kann.

Die Problematik, einen zentralisierten Zugriff auf die dezentrale Struktur der *DSPA* zu gewährleisten, stellt keineswegs eine triviale Aufgabe dar. Lösungen dieser Problematik sind vielmehr immer noch Gegenstand der Diskussion auf dem Gebiet der verteilten Systemverwaltung.

3.1.5 Datenbankanbindung

Alle Objekte, die vom Modellmanagement in irgendeiner Weise verwaltet und organisiert werden (Rechner, Simulationsprogramme in verschiedenen Versionen, die über das Datenmanagement ausgetauschten Daten, etc.), sind aus einer abstrakten Sicht heraus durch Eigenschaften bzw. Informationen repräsentiert. Ein großer Teil davon muß dauerhaft (*persistent*), über die Zeitdauer eines Simulationsvorhabens hinaus gespeichert werden können. Beispielsweise sind alle Daten, die im oben beschriebenen Schritt der Integration extrahiert werden, sogenannte *offline*-Informationen; sie werden in einem Testlauf gewonnen, müssen aber auch *online*, d.h. zur Ausführung eines Simulationsexperiments, zur Verfügung stehen.

Folglich muß in das Modellmanagement eine Datenbank und deren Verwaltung integriert werden.

3.2 Nutzergruppen

Unterstützung des Menschens als primäre Richtlinie des Modellmanagements bedeutet, daß das Design und die Umsetzung der Modellverwaltung sich am Kenntnisstand und den Fähigkeiten des Menschen orientieren muß ([Bul94]). Dazu muß jedoch erst die Frage beantwortet werden, welche Menschen mit dem Modellmanagement interagieren werden.

Eine Benutzeranalyse für die Simulationsarchitektur *DSPA* ist implizit bei der Beschreibung des Lebenszyklus eines Simulationsmodells (siehe Kapitel 2.1.3 und 2.2.2) durchgeführt worden. Als Ergebnis dieser Analyse sind große Unterschiede bezüglich Wissen und Qualifikation innerhalb des mit dem Modellmanagement interagierenden Personenklientels festzustellen. Eine Unterteilung des Klientels in Nutzergruppen erscheint daher sinnvoll, damit das Modellmanagement dem Menschen abhängig von seiner Befähigung die optimale Unterstützung zukommen lassen kann.

1. **Nutzer**

   Der Nutzer ist diejenige Person, die die Simulationsumgebung benötigt, um Versuchsreihen zur Unterstützung verschiedenster Forschungsvorhaben durchzuführen. Für ihn ist der Forschungssimulator reines Mittel zum Zweck, dessen Leistungsfähigkeit er zur Beantwortung anderer inhaltlicher Fragen abrufen möchte.

Er ist ein Anwender, der eine möglichst einfache und intuitive Bedienung des Forschungssimulators bevorzugt, und in der Regel über wenig bis gar kein Hintergrundwissen verfügt.

2. **Entwickler**

   Die Gruppe der Entwickler umfaßt die Mitarbeiter in den Forschungsvorhaben, die durch ihre Forschungsziele festlegen, welche Szenarien der Forschungssimulator wiederzugeben in der Lage sein muß. Handelt es sich hierbei um neuartige oder erweiterte Szenarien im Hinblick auf das, was der Simulator zu leisten imstande ist, dann tragen die Entwickler mit von ihnen programmierten Simulationsmodulen zur Erweiterung der Simulationsmodelle bei.

   Der Wissensstand der Entwickler umfaßt im wesentlichen detaillierte Kenntnisse über die von ihnen programmierten Simulationsmodule und einen oberflächlichen Überblick über die Struktur der verteilten Simulationsarchitektur *DSPA*.

3. **Administrator**

   Der Administrator ist derjenige, der die Simulationsarchitektur betreut und wartet. Er ist dafür zuständig, den Forschungssimulator an Änderungen aller Art (z.B. durch neue Rechnerhardware oder neue Simulationsmodule) anzupassen, ihn für die Benutzung durch andere vorzubereiten und ihn in Zusammenarbeit mit den Entwicklern aus den unterschiedlichen Forschungsprojekten weiterzuentwickeln.

   Er ist derjenige, der die verteilte Simulationsarchitektur im gesamten überblickt. Hinsichtlich der verwalteten Simulationsmodelle besitzt er nur funktionelles, aber kein inhaltliches Wissen (*„Black-Box-Ansatz“*, siehe [And00]). Ist zur Administration der *DSPA* weitergehendes Wissen über das „Innere“ eines Simulationsmoduls vonnöten, muß der entsprechende Entwickler miteinbezogen werden. Der Administrator nimmt somit eine Mittlerstellung zwischen der Gruppe der Benutzer und der Gruppe der Entwickler ein.

3.3 Konzeptioneller Rahmen

Das Modellmanagement verfolgt den Ansatz, zwischen dem Menschen und der Simulationssoftware zu vermitteln. In diesem Sinne ist es **nicht zwingend** erforderlich, daß eine Simulationsarchitektur ein Modellmanagement dieser Art beinhaltet. Das

Verwalten und Ausführen der Simulationsprogramme, verbunden durch die Schnittstelle des Datenmanagements ([Eng01]), kann theoretisch „von Hand“ erfolgen, was den Menschen aber aufgrund der komplexen und verteilten Struktur der Simulators relativ bald überfordert und dadurch Risiken in sich birgt.

Die Vorteile, und damit die Rechtfertigung für diese Arbeit, können durch folgende Schlagwörter charakterisiert werden:

- „Ease Of Use“
- Automatisierung
- Qualitätssicherung

Abstrahiert man die Sichtweise des Vermittlers, so läßt sich das Modellmanagement stark vereinfacht als

- eine Umgebung zur Ausführung von Programmen beschreiben,
- die einen gemeinsamen Zweck erfüllen,
- der vom Benutzer festgelegt worden ist.

Der konzeptionelle Rahmen für das Modellmanagement muß zuallererst die grundlegenden Randbedingungen der Simulationsarchitektur *DSPA* (siehe Abschnitt 2.2.2) berücksichtigen. Darüberhinaus lassen sich folgende Ansätze formulieren.

Systemgrenzen

Die zu verwaltende Komponente des Modellmanagements ist ein Simulationsprogramm. Die Verwaltung ist jedoch nicht festgelegt auf eine spezifische Anwendung, sondern soll vielmehr auf eine Familie bzw. Domäne derselben anwendbar sein ([Smi99], [Bäu98]). Demzufolge muß dem Modellmanagement eine verallgemeinerte, generalisierte Vorstellung eines Simulationsprogramms zugrunde gelegt werden. Der Ansatz hierfür ist schon in Kapitel 2.2.2 beschrieben worden, indem der Leistungsumfang der betrachteten Simulationsarchitektur anhand der Unterteilung in *allgemeine* und *modellspezifische Aufgaben* abgesteckt worden ist.

Die allgemeinen Aufgaben aus der Sicht der Modellverwaltung sind weiter oben beschrieben; Beispiele hierfür sind:

- Ein Simulationsprogramm benötigt zur Ausführung seiner selbst eine spezifische Computer-Hardware, z.B. aufgrund der Leistungsfähigkeit derselben. Diese soll durch das Modellmanagement bereitgestellt werden.

- Simulationsprozesse können zur Laufzeit in Konflikt geraten, wobei auf die Definition eines Konflikts hier nicht weiter eingegangen werden soll. Diese Konflikte müssen vom Modellmanagement entdeckt und behoben werden.

- Ein Simulationsprogramm kann zur sinnvollen Erfüllung seiner Simulationsaufgabe von anderen Programmen abhängen. Diese Abhängigkeiten muß das Modellmanagement kennen, so daß bei der Auswahl eines Simulationsprogramms automatisch die dazugehörigen Komponenten mit ausgewählt werden können.

Modellspezifisch oder in diesem Kontext programmspezifisch ist

1. alles, was innerhalb eines Simulationsprogramms geschieht sowie

2. alle inhaltlichen Aspekte, die sich auf die Abbildung eines realen Systems durch das Simulationsprogramm beziehen.

Ersteres meint die softwaretechnische Implementation, z.B. wie sieht die Klassen-Struktur eines Programms aus oder welchen Kontrollfluß besitzt die Anwendung? Als Beispiel für den zweiten Punkt betrachten wir die Implementation eines Triebwerks-Modells für ein simuliertes Flugzeug. Das Wissen, welches Set von Daten, den Zustand des Triebwerks wiedergibt, ist modellspezifisch. Es gilt nur für diese eine Implementation und könnte für die Repräsentation eines anderen Triebwerks schon nicht mehr gültig sein. Allgemein kann man sagen, daß Informationen, die den beiden oberen Punkten zuzuordnen sind, dem Wandel unterworfen sind. Denn gerade die Weiter- bzw. Neuentwicklung von Simulationsmodulen in der Simulationsarchitektur *DSPA* soll einfach und uneingeschränkt möglich sein. Ein auf derartigen Informationen aufgebautes Fundament für das Modellmanagement wäre keinesfalls stabil und würde in der Zukunft unweigerlich zu Problemen führen.

Als Konsequenz aus obigen Ausführungen wird ein Simulationsprogramm bzw. ein Simulationsprozeß (als Instanz des Programms zur Laufzeit) **immer** als „Black-Box“ betrachtet werden. Es ist eine unteilbare, atomare Einheit. Das Modellmanagement darf nur auf Informationen zurückgreifen, die sich aus dem Wirken eines Prozesses

nach außen ergeben. Mit „nach außen“ ist in diesem Fall das Kommunikationsverhalten, sprich der Datenaustausch mit anderen Prozessen über den „Datenpool“ ([Eng01]) gemeint. Diese Betrachtungsweise wird im allgemeinen als *funktionaler Ansatz* bezeichnet ([And00]).

Ein erwünschter Nebeneffekt des Black-Box-Ansatzes ist die Tatsache, daß aufgrund der Unkenntnis der inneren Details der Simulationsprogramme bzw. der Simulationsprozesse deren Ausführung nicht unzulässig durch das Modellmanagement beeinflußt werden kann. Das Modellmanagement soll so weit wie möglich nur als Beobachter auftreten.

Trennung von Kontrolle und Ausführung

Ein Simulationsprogramm ist die Umsetzung einer Teilaufgabe eines Simulationsmodells. In der Regel läßt sich die Art, wie die Aufgabe erfüllt werden soll, vom Menschen beeinflussen. Ein Beispiel hierfür ist eine Implementation zur Regelung der Lage und Position eines simulierten Flugzeugs. Die Lösung des Problems kann mit Hilfe verschiedener zugrundegelegter Regelgesetze erfolgen, zwischen denen möglicherweise hin- und hergeschaltet werden soll. Ein anderes Beispiel ist die Festlegung der Anfangsbedingungen für ein simuliertes Flugzeug, wie z.B. die Abflugmasse des Flugzeugs oder die Anfangsposition, von der das Flugzeug aus starten soll. Für diese Art Prozeßsteuerung, die meist auch online zur Verfügung stehen soll, muß der Entwickler eines Simulationsprogrammes sorgen, da er auch am besten über die Definition und Umsetzung der Simulationsaufgabe Bescheid weiß.

Da das Modellmanagement einen zentralen Arbeitsplatz zur Bedienung des Gesamtsimulators zur Verfügung stellen will, ist es unvermeidlich, daß die eigentliche Aufgabe eines Simulationsprogramms getrennt von der oben beschriebenen Art der Kontrolle implementiert wird. Ein jeder Simulationsprozeß besteht also aus zwei zusammengehörigen Einheiten: einen Ausführungs- und einen Kontrollpart, welche sich über die Schnittstelle des Datenmanagements austauschen.

Nur so wird eine verteilte Ausführung, der Ausführungspart auf einem beliebigen Rechner, der Kontrollpart auf dem Arbeitsplatzrechner des Benutzers, möglich. Durch diese Festlegung wird zusätzlich die Bedienung des Gesamtsimulators vereinheitlicht, sowie die Möglichkeit geschaffen, ein Simulationsmodul und dessen Bedienung möglichst entkoppelt voneinander zu entwickeln.

Die Vorgabe grober Designrichtlinien zur Gestaltung graphischer Benutzerober-

flächen für die Kontrollprozesse soll ein konsistentes „look and feel“ gewährleisten, und damit dem Menschen die Bedienung erleichtern.

Simulations- und Systemprozesse

Die Vorbereitung der Simulationsarchitektur bzw. des Rechnernetzwerkes für ein Simulationsvorhaben ist eine der oben beschriebenen Aufgaben des Modellmanagements. Dazu muß es auf die Hilfe von externen Prozessen zurückgreifen, da z.B. Fileserver- oder Datenbankdienste, die zur Simulation benötigt werden, nicht auch noch von der Modellverwaltung übernommen werden können. Die Koordination und Konfiguration, welche Prozesse auf welchen Rechnern gestartet werden sollen, kann aber mit Hilfe des Modellmanagements geschehen, da es ja nach obiger, abstrakter Darstellung eine Umgebung zur Ausführung von Programmen ist. Die Nutzung dieser Funktionalität ist aufgrund der verteilten Natur und der damit verbundenen Komplexität des Forschungssimulators auf jeden Fall anzuraten.

Unter der Begrifflichkeit *Systemprozeß* werden daraus abgeleitet alle Prozesse verstanden, die in irgendeiner Art und Weise zur Vorbereitung und dem Betrieb der Simulationsarchitektur oder dem Rechnernetzwerk benötigt werden, ohne aber einen Zusammenhang zur inhaltlichen Simulationsaufgabe zu besitzen. Sie können durch das Modellmanagement verwaltet und in die Simulationsarchitektur integriert werden. In Analogie zu den Simulationsprogrammen sind die Systemprogramme eine zweite Form von Komponenten, die vom Modellmanagement berücksichtigt werden müssen.

Es sei aber darauf hingewiesen, daß die Verwaltung der Systemprozesse sich auf die oben genannten Aspekte, nämlich auf welchen Computern soll der Prozeß zur Ausführung gebracht werden, beschränken muß. Für eine umfassendere Verwaltung fehlt dem Modellmanagement jegliche Informationsgrundlage.

Offenes System

Die Modellverwaltung und mit ihr die Simulationsarchitektur sollen ein offenes System darstellen. Die Anforderungen und Ansprüche an komplexe Systeme können nicht immer a priori erfaßt und berücksichtigt werden, unter anderem auch deswegen, weil Ansprüche sich ändern können.

Die Umsetzung des Modellmanagements soll die Beachtung dieser Leitlinie beherzigen, indem immer wieder die folgenden Fragen gestellt werden:

- Kann ein Aspekt im Laufe der Zeit Änderungen unterworfen sein („subject to change“)?
- Wie müssen eventuelle Änderungen am Modellmanagement erfolgen?
- Welche Möglichkeiten können vorgesehen werden, im Falle einer Änderung deren Aufwand minimal zu halten?

Für folgenden Bereiche kann schon jetzt vorhergesagt werden, daß im Laufe der Zeit eine Änderung oder Erweiterung sehr wahrscheinlich sein wird:

- Die Benutzerschnittstelle
- Die Verwaltung der Ressourcen des Rechnernetzwerkes
- Die Vorbereitung der Simulationsarchitektur

Abschließend soll betont werden, daß der Begriff „offenes System“ nicht dahingehend verstanden wird, daß das Modellmanagement als Programm in der Lage sein wird, auf alle geänderten Anforderungen dynamisch zu reagieren. Änderungen und Erweiterungen werden in der Regel nicht ohne Programmieraufwand möglich sein, sei es nun eine Erweiterung durch einen getrennten Systemprozeß oder eine Erweiterung des Modellmanagements direkt.

3.4 Vorgehensweise

Das Daten- und Modellmanagement sind die Organisationseinheiten der verteilten Simulationsarchitektur *DSPA*. Ihre Funktionalität bietet einen Rahmen für die Benutzung der Simulationsumgebung am Fachgebiet Flugmechanik und Regelungstechnik. Nach [Bäu98] kann man sie demzufolge als *„framework for specific domains“* charakterisieren, einer von vier gegenwärtigen Forschungsschwerpunkten in der Informatik in Bezug auf Softwarearchitekturen.

In den Verantwortungsbereich des Modellmanagements fällt die Verwaltung des Simulationsmodells. Erster Schritt zur Erfüllung der damit verbundenen Aufgaben ist eine Analyse des Simulationsmodells, welche im folgenden Kapitel (Kapitel 4) beschrieben werden soll. Ziel ist es, aus der Sicht des Modellmanagements die für den Anwendungsbereich *„Simulationsmodell“* relevanten Komponenten und deren

Beziehungen zu identifizieren. Das wird aus vier verschiedenen Blickwinkeln heraus geschehen, um die Komplexität der einzelnen Betrachtungsebenen nicht zu groß werden zu lassen ([And00]).

Das auf diese Weise erarbeitete *fachliche Modell* ([Bäu98]) stellt die Grundlage für eine softwaretechnische Umsetzung des Modellmanagements dar. Bevor diese jedoch angegangen wird, soll im Kapitel 5 ein Überblick über aktuelle Konzepte und/oder Implementationen gegeben werden, die sich mit der gleichen oder ähnlichen Problemstellungen wie die vorliegende Arbeit beschäftigen. Dabei müssen folgende Fragen beantwortet werden:

- Was ist der Gültigkeitsbereich der Anwendung bzw. des Konzepts?
- Sind die gemachten Erfahrungen für die Entwicklung des Modellmanagements nutzbar?
- Sind Teilbereiche vorhandener Lösungen für das zu entwickelnde Modellmanagement übertragbar und adaptierbar?

Das Ziel der Recherche ist, zu einer Aussage zu gelangen, wie für die Umsetzung des Modellmanagements verfahren werden soll. Die Tatsache, daß das Modellmanagement der Kategorie der *domänen-spezifischen Rahmenwerke* zugeordnet wird, läßt jedoch schon erahnen, daß es sich hier um eine für das Fachgebiet Flugmechanik und Regelungstechnik maßgeschneiderte Anwendung handeln wird. Demzufolge ist zu vermuten, daß die Recherche allenfalls im Hinblick auf anwendbare Konzepte oder generalisierte Lösungen Eingang in die Umsetzung des Modellmanagements finden wird.

4 Analyse des Simulationsmodells

Die Umsetzung eines theoretischen Simulationsmodells zur Ausführung auf einem oder mehreren Computern ist trivial formuliert nichts anderes als eine Anzahl von Simulationsprogrammen, die voneinander abhängen bzw. sich gegenseitig beeinflussen können. Ein Beispiel für die nichtsdestoweniger komplexe Struktur, die zur Laufzeit von den Simulationsprozessen aufgespannt werden kann, zeigt Abbildung 4.1.

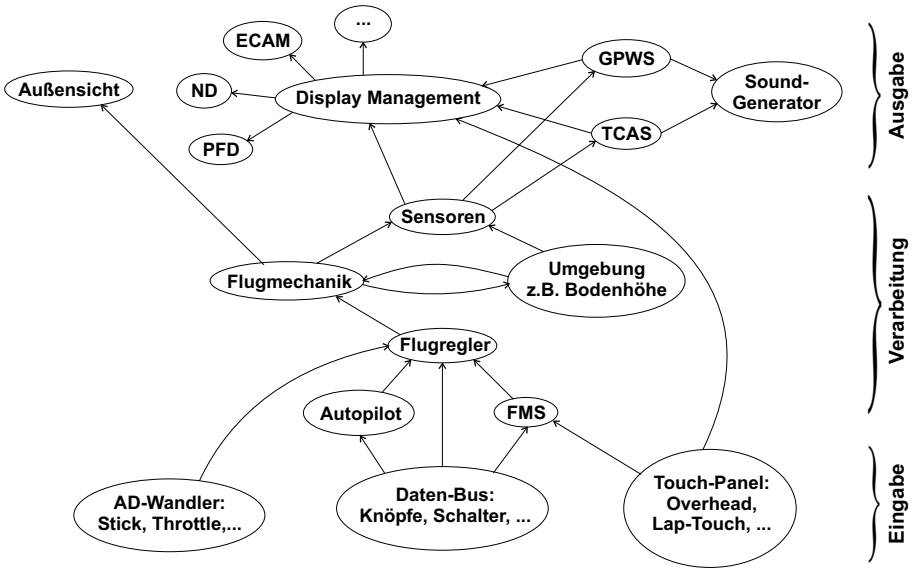


Abb. 4.1: Beispiel für Prozeßabhängigkeiten eines implementierten Simulationsmodells

Das Modellmanagement kann seine Aufgabe, die Summe der Simulationsprozesse, die ein Simulationsmodell implementieren, zur Laufzeit zu konfigurieren und zu überwachen, nur dann übernehmen, wenn es sich ein *fachliches Modell* ([Bäu98]) des Simulationsmodells erarbeitet. Dieses identifiziert, ausgehend von der vorher formulierten Aufgabenstellung, die Strukturen und Komponenten des Anwendungsbereichs. Erst durch diesen Schritt der Abbildung ist das Modellmanagement in der Lage, Aussagen zu treffen, Bewertungen vorzunehmen, Prozesse zu identifizieren, etc. ([And00]). Es wird eine Grundlage geschaffen, anhand derer ein Soll-Verhalten für Programme bzw. Prozesse definiert und mit einem aktuellen Verhalten (Ist-Verhalten) verglichen werden kann.

Um komplexe Systeme, wie beispielsweise ein zur Ausführung gebrachtes Simulationsmodell, besser überschauen zu können, wird die Analyse derselben häufig aus

verschiedenen Sichtweisen heraus durchgeführt ([And00]). Diese Methodik soll auch hier zur Erarbeitung eines fachlichen Modells für das Modellmanagement zur Anwendung kommen, indem das Simulationsmodell aus vier unterschiedlichen Blickpunkten heraus betrachtet wird:

1. Ausführungsanforderungen
2. Einzelner Prozeß
3. Prozeßstruktur
4. Datenstruktur

Für jede der vier verschiedenen Betrachtungsebenen, die in den nächsten Abschnitten im Detail erläutert werden, müssen folgende Fragen beantwortet werden:

- Welchen Aspekten gilt das Hauptaugenmerk?
- Was sind die Grenzen der Betrachtung?
- Welche Komponenten existieren und wie stehen sie in Verbindung?

4.1 Ausführungsanforderungen

Die Ausführung einer Simulation für unsere Zwecke bedeutet die Ausführung einer je nach Simulationsszenario wechselnden Anzahl von Prozessen. An die zugehörige Rechnerumgebung werden dabei individuell verschiedene Anforderungen gestellt, die im Zuge der Verwaltung von Simulationsprozessen mitberücksichtigt werden müssen (siehe Abbildung 4.2).

Dies kann jedoch nicht statisch geschehen, da erst kurz vor der Laufzeit einer Simulation festgelegt wird, auf welchen Rechnern ein Prozeß gestartet werden soll. Folglich müssen die benötigten Ressourcen eines Simulationsprozesses mitmodelliert und in das Modellmanagement integriert werden.

Gegenstand der Modellierung ist folglich die Beziehung der beiden Komponenten *Prozeß* und *Rechner*, wobei erstere Anforderungen an den Rechner stellt, welche dieser aufgrund seiner Leistungsfähigkeit zu erfüllen in der Lage sein muß. Vom Standpunkt des Prozesses aus ist in diesem Zusammenhang von Bedeutung:

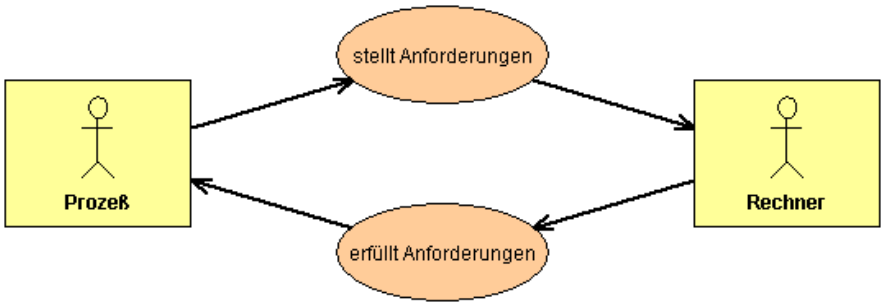


Abb. 4.2: Ausführungsanforderungen

- Welches Betriebssystem besitzt der Rechner?
- Wieviele und welche Art von CPU's sind vorhanden?
- Über wieviel Hauptspeicher verfügt der Rechner?
- Welche I/O-Bausteine sind in den Rechner integriert?

  (Grafikkarte, Festplatte, serielle Schnittstelle, ...)

Da für die Ausführung eines Simulationsprozesses die Leistungsfähigkeit der benutzten Hardware von Bedeutung ist, muß in die Modellierung der Ressourcen eine quantitative Aussage über die Leistungsfähigkeit bzw. die Belastung des Rechners miteingeschlossen werden. Dies ist beispielsweise von Interesse, wenn sich mehrere Prozesse die Hardware eines Rechners teilen sollen, und sich die Frage stellt, ob die Leistungsfähigkeit des Rechners ausreicht, die Anforderungen in ihrer Summe zu befriedigen.

Das Ziel dieser Arbeit ist nicht, eine optimale Modellierung eines Computers und seiner Bausteine zu entwickeln. Dies wäre vom Umfang her ein Unterfangen, was eine eigenständige Arbeit füllen würde. Die Verteilung der Prozesse auf die zur Verfügung stehenden Rechner als Bestandteil der Konfiguration der Simulation muß den oben beschriebenen Sachverhalt aber berücksichtigen. Demzufolge wird eine Beschreibung der Beziehung zwischen Rechner und Prozeß nur auf eine abstrakte und prinzipielle Art geschehen.

4.2 Einzelner Prozeß

Die Modellverwaltung integriert ein Programm in die Simulationsarchitektur, um es in späteren Simulationsläufen einsetzen zu können. Dabei stellt ein Programm nur einen Teil des gesamten Simulationsmodells dar, welches erst zur Laufzeit von einem Benutzer zusammengestellt wird.

Um die Aufgabe der Konfiguration der Simulation sowie die der Überwachung einer laufenden Simulation wahrnehmen zu können, muß das Modellmanagement einen Prozeß und dessen Aufgabe in der Simulation eindeutig identifizieren und charakterisieren können.

Die Abstraktion des Prozeß- bzw. Programmablaufes zeigt Abbildung 4.3:

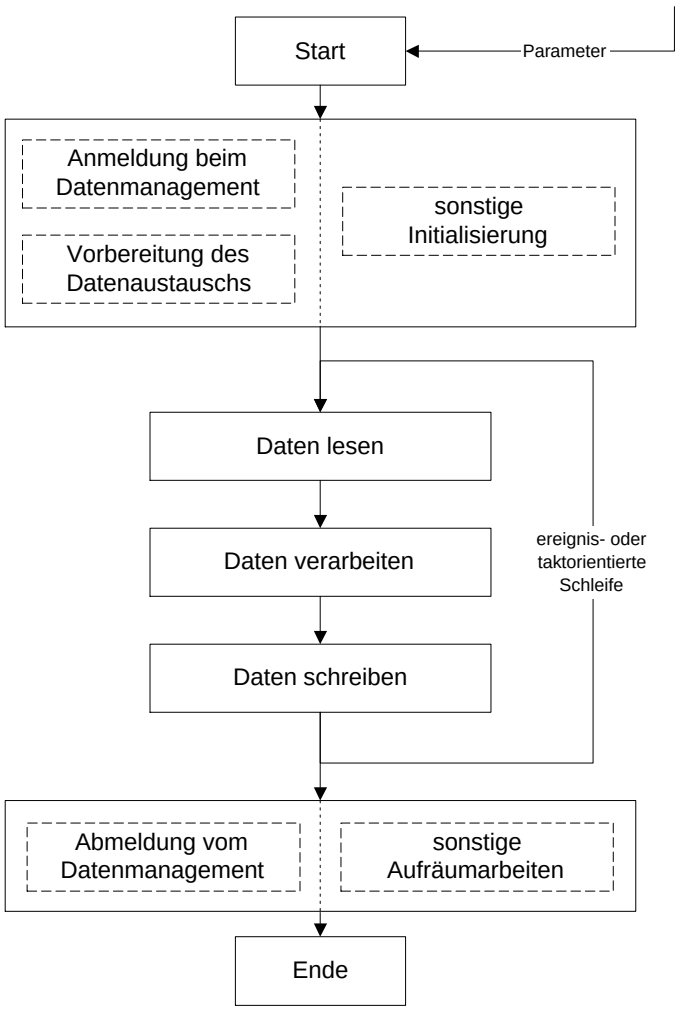


Abb. 4.3: Abstrahierter Prozeß- bzw. Programmablauf

Der Start eines Simulationsprozesses ist die Aufforderung an das Betriebssystem eines Rechners, ein bestimmtes Programm mit entsprechenden Parametern zur Ausführung zu bringen. Bestandteil der im allgemeinen folgenden Initialisierungsphase des Prozesses ist die Anmeldung desselben beim Datenmanagement und die anschließende Mitteilung, welche Daten der Prozeß zu lesen und schreiben beabsichtigt.

Nach Abschluß der Initialisierung tritt ein Prozeß in der Regel in eine ereignis- oder taktgesteuerte Schleife ein, die sich abstrakt betrachtet in drei Schritte untergliedern läßt:

1. Daten lesen,
2. Daten entsprechend der Prozeßaufgabe verarbeiten und
3. Daten schreiben.

Der Austritt aus der Schleife bedeutet das Ende des Prozesses, dem aber noch „Aufräumarbeiten“ vorausgehen, wie beispielsweise die Abmeldung des Prozesses vom Datenmanagement.

Anhand dieser Beschreibung kann man erkennen, daß die Betrachtung des einzelnen Prozesses sich mit dessen „Einbettung“ in den Datenpool beschäftigt, wobei der Datenpool die Systemgrenze für diese Sichtweise festlegt (siehe Abbildung 4.4). Der Datenpool ist die Bezeichnung für den transparenten, verteilten Speicher des Datenmanagements, in dem alle Informationen einer Simulation festgehalten werden.

Die Übertritte über die Systemgrenze des Datenpools hinweg kennzeichnen die Interaktion der Simulation mit dem Menschen, in diesem Fall mit *„IN“* und *„OUT“* bezeichnet.

Eingänge sind Aktionen, Informationen, Anweisungen, etc. des Menschen, die in eine für die Simulationsprozesse verwendbare, digitale Form gebracht und über eine Art „Gateway“-Prozeß in den Datenpool geschrieben werden. Von dem Zeitpunkt an sind die entsprechenden Informationen für alle an der Simulation teilnehmenden Prozesse verfügbar. Ein Beispiel hierfür ist die Betätigung der primären Steuerelemente im Forschungscockpit durch einen Piloten. Unter anderem die Auslenkung des Sidesticks oder die Stellung der Schubhebel werden durch Meßeinrichtungen analog erfaßt, durch Wandlerkarten in eine digitale Form überführt und dann von einem Simulationsprozeß an den Datenpool geschickt.

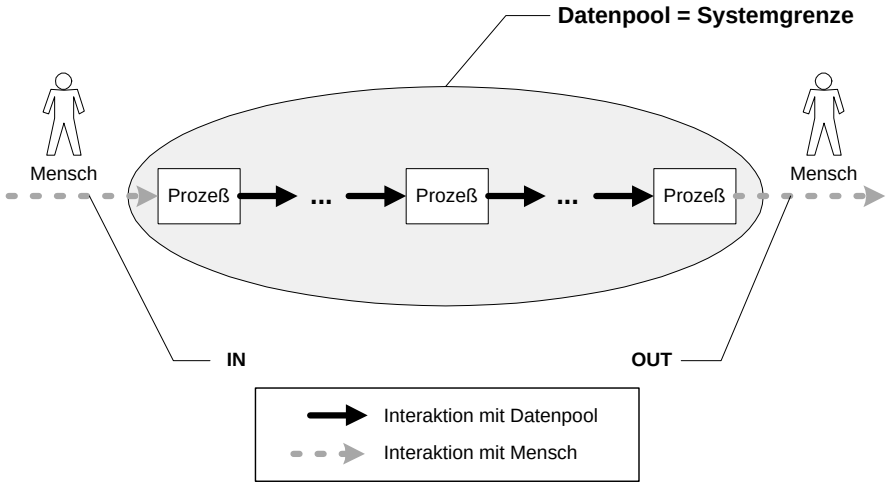


Abb. 4.4: Sichtweise des einzelnen Prozesses

Das entsprechende Gegenstück, die Ausgänge, umfaßt die Aufbereitung digitaler Information bzw. Zustände in eine für den Menschen geeignete Form. Die Flugführungsdisplays des Fachgebiets Flugmechanik und Regelungstechnik sind hier als Beispiel anzuführen. Diese erzeugen mit Hilfe der Information über Position und Lage des simulierten Flugzeugs eine dreidimensionale Voraus- und Draufsicht, die den Piloten auf den vor ihnen befindlichen LC-Displays dargestellt werden.

Die Prozesse in ihrer Interaktion mit dem Datenpool können in drei abstrakte Kategorien eingeteilt werden (siehe Abbildung 4.5): *PROCESSOR*, *SINK* und *SOURCE*.

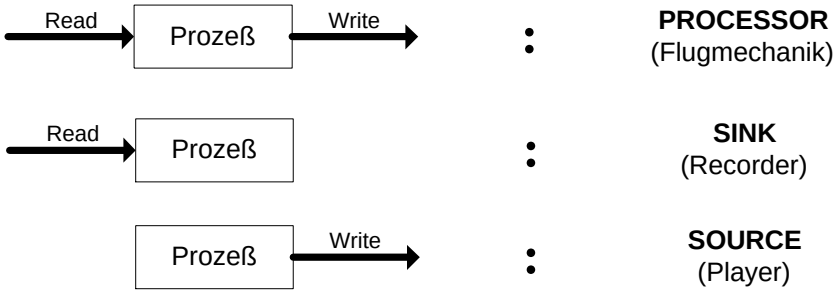


Abb. 4.5: Prozeßabstraktionen

Die erste Kategorie stellt die allgemeine Form eines Simulationsprozesses dar, er liest Daten ein, verarbeitet diese seiner Aufgabe entsprechend (*„Data Processing“*) und schreibt die Ergebnisse seiner Berechnungen in den Datenpool. In diese Kate-

gorie gehört der Prozeß, der die Starrkörper-Bewegung für das simulierte Flugzeug berechnet. Über den Datenpool werden Steuerkommandos des Piloten oder des automatischen Flugssystems zur Verfügung gestellt, welche dann Eingang in die numerische Integration der Bewegungsgleichungen eines Flugzeuges finden. Resultat der Integration ist eine veränderte Position und Lage des Flugzeuges, die von jedem anderen Prozeß in der Simulation abgefragt werden können, sobald sie dem Datenmanagement mitgeteilt worden sind.

Die zweite Form von Prozessen liest nur Daten ein und stellt somit eine Datensenke („Sink“) dar, die nicht auf die Simulation zurückwirken kann. Ein Beispiel hierfür ist ein Datenrecorder, der den Verlauf eines simulierten Fluges über der Zeit aufzeichnet, um ihn später wieder abzuspielen oder die gespeicherten Daten *offline* zu analysieren.

Das entsprechende Gegenstück, die Quelle oder „Source“, produziert Daten für die Simulation, und zwar unbeeinflußt vom aktuellen Zustand der Simulation, der durch die Summe aller Informationen im Datenpool beschrieben wird. Dieser Sparte kann beispielsweise ein Verkehrsgenerator zugerechnet werden. Er erzeugt automatisch, mit Hilfe von vereinfachten Modellen Fremdflugzeuge für das eigentliche, simulierte Flugzeug, um dem Piloten im Forschungscockpit ein realitätsgetreues Abbild der Wirklichkeit zu bieten.

Grundlage aller Abstraktionen ist eine funktionale Sicht der Prozesse, wie sie im konzeptionellen Rahmen des Modellmanagements gefordert worden ist (vergleiche Kapitel 3.3). Dabei wird ein Programm bzw. ein Prozeß vom Modellmanagement als atomare Einheit angesehen, das sich durch seine Ein- und Ausgänge auszeichnet ([And00]). Unter atomar ist in diesem Falle zu verstehen, daß eine Komponente nicht weiter aufgelöst, sondern nur als eine Black-Box betrachtet werden kann. Wissen über die *„innere Aufgabe“* eines Simulationsmoduls liegt somit nicht vor. Da die Ein- und Ausgänge jedoch für jedes Programm oder jeden Prozeß charakteristisch sind, kann das Modellmanagement durch Beobachtung derselben in die Lage versetzt werden, einen Simulationsprozeß zu identifizieren und aus seiner Sicht heraus auch zu bewerten.

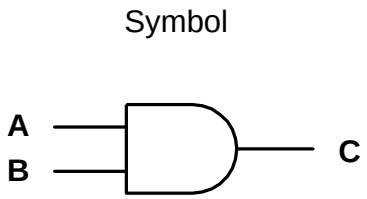
Der Datenaustausch eines Prozesses mit dem Datenmanagement stellt aus der Sicht des Modellmanagements die Aufgabe eines Simulationsprozesses dar, und ist damit vollständig von der eigentlichen, inhaltlichen Aufgabe desselben entkoppelt:

> Ein Prozeß ist aus der Sicht des Modellmanagements nichts anderes als ein mit dem Datenmanagement interagierender **Datenprozessor**, wobei die **Anzahl und Art der Daten** ihn eindeutig identifiziert.

Mit Hilfe dieser abstrakten Vorstellung kann man eine Signatur eines Prozesses definieren, auf deren Grundlage eine Entscheidung darüber getroffen werden kann,

- welche Aufgabe einem bestimmten Prozeß zuzuordnen ist und
- ob er hinsichtlich dieser ein Fehlverhalten aufweist.

Der Begriff der Signatur kommt aus der Mathematik/Informatik und beschreibt das Ein- und Ausgangsverhalten von Objekten ([NIS]). Zur Verdeutlichung des Begriffs soll der logische Operator *„AND“* dienen (siehe Abbildung 4.6; aus [Kub96]).



Wahrheitstabelle **A ∧ B**

A / B	0	1
0	0	0
1	0	1

Abb. 4.6: Der logische AND-Operator

Für diesen ist festgelegt, daß zwei Inputs zu einem Output verknüpft werden, wobei alle drei Größen nur die Werte *0* und *1* annehmen können. Die möglichen Kombinationen der Eingänge und die sich daraus ergebenden Ausgänge sind der sogenannten *Wahrheitstabelle* in obiger Abbildung zu entnehmen. Genausogut läßt sich aus der Information über Ein- und Ausgänge eine Aussage treffen, um welche Art logischen Operator es sich handeln muß.

Die Tatsache, daß die Prozeßaufgabe an dem Kommunikationsverhalten des Prozesses festgemacht wird, impliziert, daß eine Schnittstelle zum Datenmanagement vorhanden sein wird, um die ausgetauschten Daten eines Prozesses erfragen zu können.

Es soll an dieser Stelle noch einmal betont werden, daß durch das Zugrundelegen des Kommunikationsverhaltens zur Aufgabendefinition die Allgemeingültigkeit der Simulationsarchitektur gewahrt wird; es muß kein Wissen über das Innere eines

Prozesses vorhanden sein, sondern nur Kenntnisse über die Art und Weise des Datenaustauschs, welcher durch das Datenmanagement allgemeingültig für die Simulationsarchitektur vorgegeben wird.

In der Beschreibung zum abstrahierten Prozeßablauf (s.o.) ist darauf hingewiesen worden, daß ein Prozeß beim Start Parameter übergeben bekommen kann. Eine weitere Möglichkeit, den Ablauf eines Prozesses zu beeinflussen, stellen die zu jedem Prozeß gehörigen graphisch-interaktiven Kontrolloberflächen dar, die im Abschnitt 3.3 erwähnt worden sind. Beide Eingriffsmöglichkeiten können sich auf die Menge der vom Prozeß bearbeiteten Daten auswirken und müssen daher vom Modellmanagement berücksichtigt werden. Ohne Informationen über die Inhalte dieser Parameter wird das Kommunikationsverhalten der Prozesse nur unvollständig erfaßt werden können, was möglicherweise die Aufgabenerfüllung des Modellmanagements beeinträchtigt.

Die Bestimmung der Signatur findet während des „Eincheckvorgangs“ des Prozesses statt (vergleiche Kapitel 3.1.1), welcher relativ kurz ist. Deswegen können durchaus noch Fehler in der Charakterisierung eines Prozesses enthalten sein. Es ergibt sich also die Notwendigkeit, von Zeit zu Zeit, während eines Simulationslaufes, die Signatur eines Prozesses zu überprüfen, um iterativ die Beschreibung des Kommunikationsverhaltens zu verbessern.

4.3 Prozeßstruktur

Mit der abstrakten Prozeßaufgabe können singuläre Aussagen über das Verhalten eines einzelnen Prozesses gemacht werden. Die Gesamtsimulation ergibt sich aber durch das Zusammenspiel aller Prozesse und wird vom Menschen auch nur in dieser Dimension wahrgenommen. Demzufolge muß das Modellmanagement ebenfalls in der Lage sein, die Simulation aus einer ganzheitlichen Betrachtung, die alle Prozesse miteinschließt, zu beurteilen.

Ein Beispiel für eine ganzheitliche Sichtweise gibt Abbildung 4.7, in der sich die Abhängigkeiten der einzelnen Prozesse erkennen lassen. In dieser Darstellung ist der Datenpool aus der vorherigen Betrachtungsebene transparent und die Beziehungen der Prozesse untereinander ergeben sich aus den ausgetauschten Daten. Diese Verbindungen sind gedachter Natur, da das Datenmanagement alle Prozesse voneinander entkoppelt und keine direkte Kommunikation zwischen den Prozessen zuläßt.

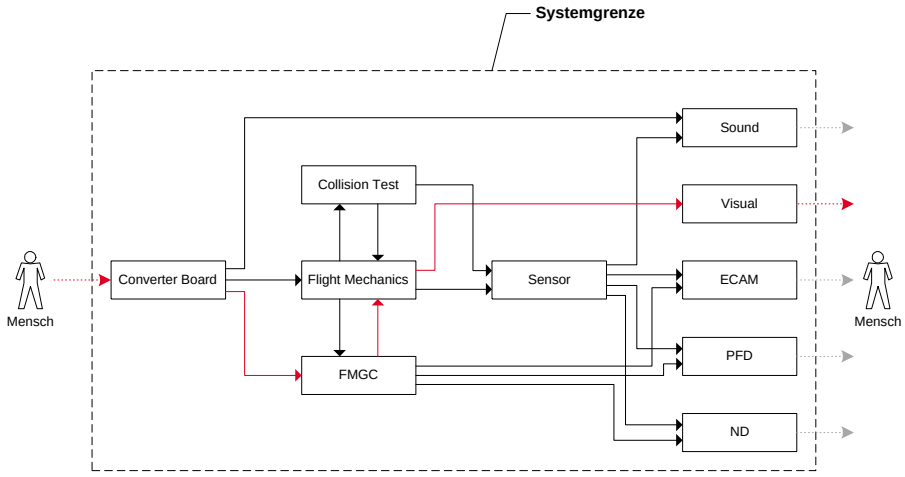


Abb. 4.7: Prozeßstruktur

Für den Menschen als der mit der Simulation interagierende Part ist der *„rote Faden“* (siehe Abbildung 4.7) von Bedeutung. Ein Steuerkommando seinerseits wird vom dafür zuständigen Prozeß, dem *Converter Board*, in eine digitale Darstellung gewandelt und in den Datenpool geschrieben. Die skalierte Information wird vom *Flight Management Guidance Computer* des simulierten Flugzeugs interpretiert und in entsprechende Ausschläge für die Steuerflächen (Ruder) des Flugzeugs transformiert. Diese finden Eingang in die numerische Integration der Bewegungsgleichungen des Flugzeugs (*Flightmechanics*). Die veränderte Lage und Position des Flugzeugs wird vom Menschen mit Hilfe der Darstellung seiner Sicht nach außen (*Visual*) wahrgenommen, so daß sich der Kreis zum Menschen wieder geschlossen hat.

Gesetzt der Fall, der *Flight Management Guidance Computer* würde in dieser Kette nicht vorhanden sein. Dann würden die anderen drei Prozesse für sich betrachtet fehlerfrei arbeiten, die reale Situation würde dem Menschen aber völlig falsch wiedergegeben werden, da das Flugzeug nicht den Eingaben des Piloten folgt. In diesem Zusammenhang würde man aus der Sicht des Menschen eine fehlerhafte Implementation der Starrkörperdynamik des Flugzeugs (*Flightmechanics*) vermuten, die wahre Ursache ist jedoch, daß die Kommandos des Piloten nicht bei dem betreffenden Prozeß ankommen, was sich für diesen so darstellt, als ob gar keine Eingaben gemacht worden sind.

Die Beurteilung der Prozeßstruktur unterliegt den gleichen Randbedingungen wie die Betrachtung des einzelnen Prozesses, auch hier ist jeder Prozeß als Black-Box zu betrachten, so daß auf Informationen über die inhaltliche Aufgabe eines Simula-

tionsprozesses nicht zurückgegriffen werden kann. Als Konsequenz bleibt einzig und allein die Analyse der Datenflüsse zwischen den Prozessen, und zwar im Hinblick auf folgende Fragestellungen:

1. Welche Eingaben durch den Menschen (*„IN“*) können sich auf welche Rückmeldungen an den Menschen (*„OUT“*) auswirken?
2. Welcher Datenfluß ist mit einer Eingabe und einer korrespondierenden Rückmeldung an den Menschen verbunden?
3. Gibt es im Rahmen der Datenflüsse kollidierende Informationen?
4. Gibt es unterbrochene Datenflüsse?

Die ersten zwei Fragen sind schon im obigen Beispiel angesprochen worden. Der Sachverhalt, daß Informationen kollidieren, ist immer dann gegeben, wenn zwei Prozesse ein und dieselbe Information in den Datenpool schreiben wollen. Weder der Datenpool noch Prozesse, die diese Information aus dem Datenpool lesen, sind in so einer Situation in der Lage zu unterscheiden, von welchem Prozeß eine gelesenes Datum stammt bzw. welcher Prozeß die Priorität hinsichtlich der Information besitzen soll. Diese Konstellation ist in Abbildung 4.8 wiedergegeben.

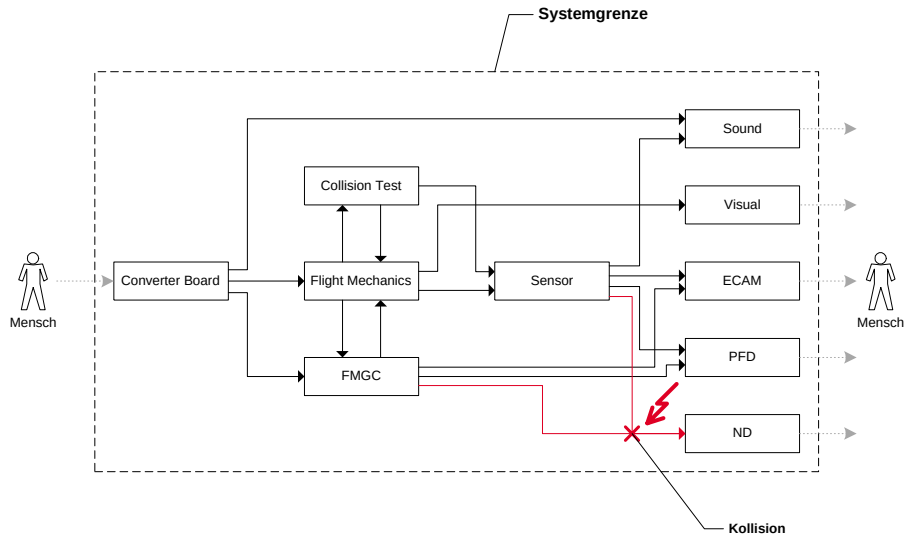


Abb. 4.8: Kollision von Informationen

In Konsequenz zu den gemachten Erläuterungen postuliert das Datenmanagement die Eindeutigkeit des Ursprungs einer Information als einen seiner zentralen Grundsätze ([Eng01]) und bewertet eine Verletzung desselben als Fehler.

Der Frage nach den unterbrochenen Datenflüssen ist ebenfalls schon im einführenden Beispiel erläutert worden und wird nur noch einmal in Abbildung 4.9 bildlich dargestellt.

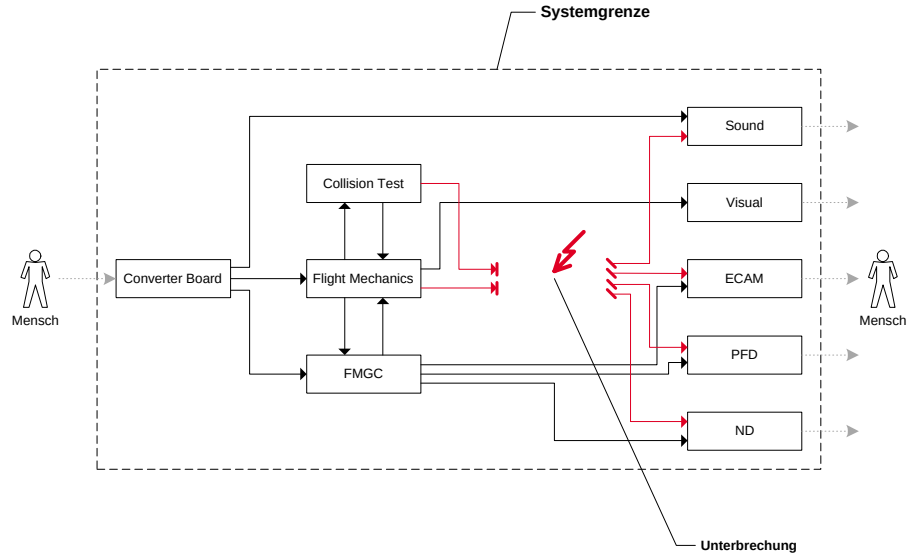


Abb. 4.9: Unterbrechung von Datenflüssen

Auf der Analyse der Datenflüsse kann dann inhaltlich aufgesetzt werden, um weitergehende Fragen des Modellmanagements zu beantworten. Diese betreffen die Zusammenstellung von Prozessen für eine Simulation als auch die Verteilung der Prozesse auf den Rechnern im Netzwerk.

Während der Konfiguration eines Simulationsvorhabens muß der Benutzer bei der Auswahl von Prozessen dahingehend unterstützt werden, so daß er ermessen kann, welche Simulationsprozesse eine inhaltliche Einheit bilden. Das kann, wie oben beschrieben, sich auf das ganze Simulationsmodell oder auch nur auf Teilbereiche desselben erstrecken. Aufgrund der Analyse der Datenflüsse ist das Modellmanagement in der Lage, ihm mitzuteilen, welche Prozesse zusammen gestartet werden sollen oder sogar müssen bzw. welche Prozesse auf keinen Fall miteinander kombiniert werden dürfen, da sie Kollisionen in den Datenflüssen erzeugen würden. Diese Prozeßgruppen können dann auch mit Hilfe des Modellmanagements gespeichert werden, so daß bei nachfolgenden Simulationen auf vorkonfigurierte Einheiten, die schon überprüft worden sind, zurückgegriffen werden kann.

Desweiteren kann die Analyse der Datenflüsse für eine optimale Verteilung der Prozesse auf den zur Verfügung stehenden Rechnern genutzt werden. Motivation einer

solchen Optimierung ist die Performance der Gesamtsimulation, die von wesentlicher Bedeutung dafür ist, daß dem Menschen eine realitätsgetreue Umgebung geboten wird, mit der er zum Zwecke von Untersuchungen interagiert.

Kritische Faktoren für die Performance sind

- die Leistungsfähigkeit der verwendeten Rechner und
- das Datenmanagement.

Der erste Fall ist dann gegeben, wenn ein Prozeß hohe Anforderungen hinsichtlich seiner Ausführung hat (siehe Abschnitt 4.1), aber auf einem ungenügend ausgestatteten Computer plaziert wird. Die unzureichende Leistungsfähigkeit des Rechners wird dazu führen, daß der Prozeß seine Aufgabe nicht mehr echtzeitnah erfüllen kann, und damit alle aufgrund der Datenflüsse von ihm abhängigen Prozesse ebenfalls beeinträchtigt werden. Da die Berechnungen eines Prozesses jedoch auf der Diskretisierung von kontinuierlichen Vorgängen basieren, müssen dessen Anforderungen hinsichtlich einer zeitlichen Taktung (Berechnungsschritte pro Sekunde) unbedingt eingehalten werden. Ansonsten wird das simulierte System, im Gegensatz zum realen, instabil und kann vom Menschen nicht mehr kontrolliert werden.

Im Falle einer Synchronisation und der Einführung einer logischen Zeit, an der sich alle Prozesse orientieren, kann eine Instabilität vermieden werden. Dann müssen aber alle abhängigen Prozesse mit der Ausführung ihrer Berechnungen warten, bis der betreffende Prozeß seinen Arbeitsschritt beendet hat. In der Konsequenz führt das dann dazu, daß der Zeitfortschritt nicht mehr mit dem vom Menschen erwarteten, „realen“ Zeitfortschritt übereinstimmt („Zeitlupe“). Somit ist eine Diskrepanz zwischen realen System und der simulierten Welt vorhanden, und die Ergebnisse eines Simulationsexperiments sind nicht mehr auf die Realität übertragbar.

Die Lösung für dieses Problem wäre die Bereitstellung eines leistungsfähigeren Rechners. Falls dies nicht möglich ist, kann aufgrund der Analyse der Datenflüsse eventuell ein Prozeß gefunden werden, der aufgrund der von ihm produzierten Daten keinen oder nur wenig andere Prozesse beeinflußt. Wird dieser dann auf dem Computer geringerer Leistungsfähigkeit plaziert, hat das weniger Auswirkungen auf die Gesamtsimulation als vorher. Die Performance kann dadurch meist so stark verbessert werden, daß die Simulation für das gewünschte Experiment wieder verwendbar ist. In der Regel ist also nicht die Leistungsfähigkeit der Rechner das eigentliche

Problem, sondern die richtige, d.h. optimale Verteilung der Prozesse auf den zur Verfügung stehenden Computern.

Der zweite limitierende Faktor ist das Datenmanagement. Dazu soll kurz die Problematik von Totzeiten erläutert werden, welche für die Beurteilung der Güte einer interaktiven Simulation (d.h. Einbeziehung des Menschen) eine wesentliche Rolle spielt.

Der Mensch in der virtuellen Umgebung der Simulation bewertet die Performance des geschlossenen Kreises, welcher dadurch beschrieben werden kann, daß eine von ihm gemachte Eingabe eine Reaktion der Simulation hervorruft, die an den Menschen zurückgemeldet wird (s.o.). Der Datenpfad, der von der Eingabe des Menschen bis hin zur Rückmeldung an denselben durchlaufen wird, kostet Zeit, einerseits die Zeit, die die Prozesse für ihre Berechnungen brauchen, andererseits die Zeit, die das Datenmanagement benötigt, eine Information von einem zum anderen Prozeß zu transportieren.

Die Summe dieser Teilzeiten, auch als Totzeit bezeichnet, kann sich ebenso fatal auf die Simulation auswirken wie die unzureichende Leistungsfähigkeit von Rechnern (s.o.). Durch zu große Totzeiten wird es unmöglich, die gewünschte, zeitliche Taktung der Prozesse zu gewährleisten, so daß eine echtzeitnahe, interaktive Simulation nicht mehr möglich ist.

Durch die Analyse des Datenaustauschs zwischen den Prozessen können die für die Wahrnehmung des Menschen kritischen Pfade optimiert werden. Das hieße beispielsweise, daß Prozesse, die intensiv miteinander kommunizieren und für die Simulation eine zentrale Rolle spielen, im Bedarfsfall auf einem Rechner plaziert werden. Die Zeit für die Übertragung einer Information zwischen zwei Prozessen, die sich auf dem gleichen Rechner befinden, ist im Vergleich zum Transport über eine Netzwerkverbindung hinweg verschwindend gering. Dadurch wird gegebenenfalls die Zeit zwischen der Aktion des Menschen und der Wahrnehmung der entsprechenden Reaktion des simulierten Systems so weit reduziert, daß man der Simulation wieder ein echtzeitnahes Verhalten attestieren und sie für das gewünschte Experiment einsetzen kann.

Einer qualitativen und quantitativen Erfassung der Totzeiten des Datenmanagements wird in [Eng01] nachgegangen. Dort werden diese Totzeiten jedoch nur gemessen und keine Gegenmaßnahmen bei Überschreitung von unzulässigen Grenzwerten unternommen.

4.4 Datenstruktur

Diese Ebene beinhaltet eine ganzheitliche Betrachtung des Simulationsmodells, die sich rein daran orientiert, welche Informationen den Gesamtzustand der Simulation beschreiben. Die Daten werden von den Prozessen, die sie produzieren und verarbeiten, getrennt und im Datenbaum der Simulation übersichtlich und hierarchisch geordnet (siehe Abbildung 4.10).

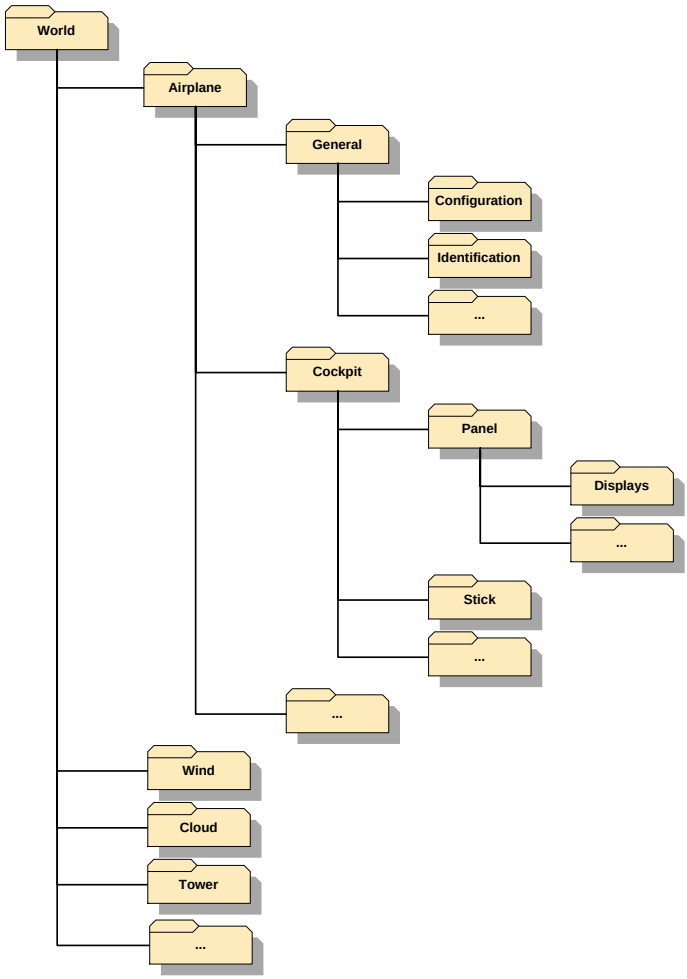


Abb. 4.10: Der Datenbaum der Simulation

Der Datenbaum ist sowohl während der Simulation als auch *offline* zu Analyse- oder Dokumentationszwecken von Interesse.

Anhand der Baumdarstellung kann man nachvollziehen, wie das Simulationsmodell in Teilaufgaben untergliedert worden ist und welche Informationen diese Teilaufgaben charakterisieren. Es läßt sich dann relativ leicht erkennen,

- ob in der Repräsentation einer Teilaufgabe Informationen fehlen,
- ob Informationen fehlerhaft sind (z.B. hinsichtlich ihres Datentyps),
- ob Informationen nicht verwendet werden,
- ob es sich um inoffizielle Informationen handelt

  (*inoffiziell* heißt in diesem Zusammenhang, daß ein Entwickler die ausgetauschten Daten eines von ihm entwickelten Simulationsmoduls nicht dem Modellmanagement und damit den anderen Entwicklern bekannt gemacht hat).

Eine automatisierte Analyse des Datenbaums kann das Modellmanagement nicht zur Verfügung stellen. Dies würde den Grundsatz, daß auf keine modellspezifischen Informationen zur Erfüllung der Aufgaben zurückgegriffen werden darf (vergleiche Kapitel 3.3), verletzen. Die Fragen, die aus dieser Sicht an das Simulationsmodell gestellt werden, können deswegen nur in Zusammenarbeit mit dem Menschen, in diesem Falle mit der Gruppe der Entwickler sowie dem Administrator, beantwortet werden.

5 Recherche bestehender Konzepte und Implementationen

Eine Simulation in der verteilten Simulationsarchitektur *DSPA* ist dadurch gekennzeichnet, daß mehrere Komponenten zur Ausführung in einem Rechnernetzwerk miteinander kombiniert werden. Jede einzelne Komponente erfüllt eine klar definierte, abgetrennte Teilaufgabe in der Simulation und wird unabhängig von allen anderen implementiert. Auf diese Weise sind maßgeschneiderte Lösungen für Simulationsexperimente möglich, nichtsdestoweniger können bereits entwickelte Simulationsmodule in neuen oder anderen Simulationsvorhaben wiederverwendet werden. Diese Vorgehensweise, in der Literatur mit dem Terminus „komponentenbasierte Softwareentwicklung“ belegt ([Gri99]), vermeidet Mehrfach-Entwicklungen von Software und reduziert dadurch den für ein Projekt benötigten Aufwand an Zeit und Geld.

Die Definition einer Komponente oder eines Moduls ist in der Fachwelt unterschiedlichen Sichtweisen unterworfen. Für die Simulationsumgebung am Fachgebiet Flugmechanik und Regelungstechnik wird deswegen festgehalten, daß eine Komponente aus ausführbaren Binärcode besteht und so verwendet werden muß, wie sie ist. Die starke Entkopplung sowohl der Komponenten vom Gesamtzusammenhang der Simulation als auch der Entwickler untereinander erfordert ein Gerüst (*„Skelett“*), welches die einzelnen Module koordiniert und zu einer sinnvollen Gesamtanwendung integriert. Diese Aufgabe wird von der verteilten Simulationsarchitektur *DSPA* mit ihren Organisationseinheiten *Daten-* und *Modellmanagement* übernommen.

Dem Modellmanagement kommt in diesem Zusammenhang im wesentlichen die Aufgabe der Verwaltung und Organisation der Simulationsprogramme bzw. -prozesse zu. Hierbei verfolgt das Modellmanagement in einer ersten Iteration einen sogenannten Black-Box-Ansatz, d.h. das Modellmanagement besitzt kein Wissen über das Innere eines Simulationsprogrammes, sondern nur über den von außen sichtbaren Datenaustausch (Ein- und Ausgänge) einer Komponente, welcher mit Hilfe der Schnittstelle zum Datenpool realisiert wird.

Dieser formalistische Ansatz beruht auf der Annahme, daß Benutzer und Entwickler ausreichende Kenntnisse über die zu simulierenden, dynamischen Systeme aufweisen. Hintergrund hierbei ist die Tatsache, daß die Diskretisierung der Abbildung von kontinuierlichen Systemen in Form von Computer-Programmen von der Dynamik und dem Zeitverhalten des betreffenden Systems abhängt (Stichwort „Timing“). Dieses muß bei Ausführung einer jeden einzelnen Komponente als auch bei der

Kombination aller Komponenten beachtet werden.

Die verteilte Simulationsarchitektur *DSPA* und mit ihr das Daten- und Modellmanagement streben demzufolge eine integrale Herangehensweise für den Aufbau und den Betrieb einer verteilten Simulation an, wobei eine Arbeitsteilung dahingehend besteht, daß der Mensch (Entwickler, Benutzer und Administrator) Systemwissen in die Simulation miteinbringt und die Umsetzung und Anwendung desselben durch das Daten- und Modellmanagement unterstützt werden.

Dieses Kapitel versucht nun einen Überblick darüber zu geben, wer sich ebenfalls mit dieser Thematik oder Teilaspekten davon beschäftigt. Von der konzeptionellen Seite her können die aktuellen Bemühungen in drei große Bereiche unterschieden werden: Betriebssysteme, Komponentenmodelle und die High Level Architecture. Folglich wird zuerst kurz darauf eingegangen werden, wie ganz allgemein Prozesse von Betriebssystemen gesehen werden und welche Verwaltungsaufgaben Betriebssysteme übernehmen. Mit eines der entscheidenden Merkmale von Betriebssystemen ist die Tatsache, daß ein Prozeß weitestgehend singulär, ohne Bezug zu anderen Prozessen betrachtet wird.

Eine Erweiterung in der Funktionalität bieten die sogenannten Komponentenmodelle; neben der Möglichkeit einer kontextabhängigen Charakterisierung der einzelnen, unabhängigen Komponenten wird zusätzlich festgelegt, wie die Einheiten zu einer sinnvollen Gesamtanwendung kombiniert und zur Ausführung gebracht werden müssen.

Als letztes wird die *High Level Architecture* (HLA) des Department of Defense (DoD) der USA vorgestellt werden; deren Infrastruktur ist speziell auf die Belange von verteilten Simulationen abgestimmt worden und bietet die Möglichkeit, eigenständige Simulatoren oder Bestandteile davon in einem gemeinsamen Simulationsvorhaben miteinander agieren zu lassen.

Im Anschluß daran wird exemplarisch eine Reihe von Anwendungsbeispielen beschrieben, wo die Theorie in die Praxis umgesetzt worden ist. Die Recherche schließt mit einer Diskussion der Vor- und Nachteile der jeweiligen Konzepte aus der Sicht des Modellmanagements. Als Ergebnis kann vorweggenommen werden, daß der Ansatz der Komponentenmodelle als die vielversprechendste Möglichkeit beurteilt wird. Auf der Grundlage der allgemeinen Modelle wird deswegen in dieser Arbeit ein eigenständiges Komponentenmodell formuliert werden.

5.1 Prozeßverwaltung in Betriebssystemen

Betriebssysteme steuern und überwachen die Abwicklung von Programmen auf einem Digitalrechner ([Kub96]). Sie kapseln die Hardware des Computers, so daß zu deren Benutzung eine unabhängige Schnittstelle zur Verfügung gestellt werden kann, und bilden damit die Basis für die Verwendung des Rechners durch Programme. Betriebssysteme stellen den untersten Level hinsichtlich Programm- bzw. Prozeßverwaltung dar.

5.1.1 Aufgaben

Im allgemeinen können die Aufgaben eines Betriebssystems folgendermaßen zusammengefaßt werden ([Kub96]):

1. Verwaltung von Ressourcen

   Prozesse greifen während ihrer Ausführung auf die Hardware des Rechners zu, beispielsweise auf die zentrale Recheneinheit (CPU), auf den Hauptspeicher oder auch auf eine Festplatte. Die Gewährung des Zugriffs der Prozesse sowie die Regelung, daß jederzeit nur ein Prozeß eine Ressource benutzt, umfassen die Verwaltung der Ressourcen eines Rechners durch das Betriebssystem.

2. Ausführungsreihenfolge

   Wird mehr als ein Prozeß beim Betriebssystem zur Ausführung angemeldet, so muß dieses eine Auswahl treffen, welchem der Prozesse der Prozessor zugewiesen werden soll. Dieser wird als aktiver oder *„running“* Prozeß bezeichnet. Das Spektrum der Abarbeitungsreihenfolge reicht von sequentiell (alle Prozesse werden nacheinander ausgeführt) bis hin zum Multi-Tasking, wo zeit- und prioritätsgesteuert zwischen den Prozessen hin- und hergeschaltet wird. Bei Mehrprozessor-Maschinen können eine entsprechende Anzahl von Prozessen gleichzeitig aktiv sein.

3. Anbieten zentraler Dienste

   Bei der Benutzung eines Rechners werden bestimmte Dienste so häufig benötigt, daß diese in das Betriebssystem integriert worden sind. In diese Kategorie fallen beispielsweise das Lesen und Schreiben von Dateien, Kommunikation über das Netzwerk, das Ermöglichen von Postverkehr (*„Mail“*), etc..

4. Identifikation und Authorisierung

   Der Zugriff auf die Hardware eines Rechners wie auch die Nutzung der Dienste des Betriebssystems wird nicht jedem Benutzer gleichberechtigt gewährt. In diesem Zusammenhang muß ein Betriebssystem in der Lage sein, die Identität eines Nutzers bzw. eines Prozesses festzustellen, um ihm dann unter Berücksichtigung seiner Berechtigungen einen Vorgang zu erlauben oder zu verweigern.

5.1.2 Bewertung

Für das Modellmanagement sind die Punkte eins und vier von Interesse, wohingegen die anderen beiden nicht in dessen Aufgabenbereich fallen.

Die Verteilung der Ressourcen durch das Betriebssystem erfolgt einzig und allein als Reaktion auf die Anfrage eines Prozesses. Das Betriebssystem besitzt keine Information darüber, welche Leistungsfähigkeit der Hardware von einem Prozeß erwartet wird, so daß dieser seine Aufgabe zufriedenstellend erfüllen kann. Im Hinblick auf die Gesamtperformance der Simulation ist jedoch eine optimale Ausnutzung der Ressourcen im Netzwerk von großer Bedeutung. Demzufolge scheidet eine Nutzung des Betriebssystems durch das Modellmanagement in diesem Zusammenhang aus.

Das Modellmanagement muß ebenso wie das Betriebssystem in der Lage sein, einen Prozeß eindeutig zu identifizieren. Während ein Betriebssystem jedoch nur die Prozesse unterscheidet, die lokal auf dem Rechner laufen, muß das Modellmanagement die Prozesse netzwerkweit auseinanderhalten können. Dazu kommt, daß die Identifikation seitens des Betriebssystems meist auf der Nutzerkennung und einer Numerierung der Prozesse basiert, wovon sich keinerlei Bezug zur Aufgabe des Prozesses in der Simulation ableiten läßt.

Die Betrachtung von einem mehr konzeptionellen Standpunkt aus führt ebenso zu dem Ergebnis, daß das Betriebssystem eines Rechners nicht für die Belange des Modellmanagements verwendbar ist.

Erstens existiert ein Prozeß für das Betriebssystem nur aus einer singulären Sichtweise . Daß eine Gruppe von Prozessen wie in der Simulation eine inhaltliche Einheit bildet und auch als solche betrachtet werden muß, kann und wird vom Betriebssystem nicht berücksichtigt, allein schon deswegen, weil sich die Prozeßgruppe auf mehrere Rechner verteilen kann.

Zweitens scheidet zumindest eine direkte Nutzung des Betriebssystems deshalb aus, weil für das Modellmanagement eine Plattformunabhängigkeit angestrebt wird, um den Einsatz des Modellmanagements in dem heterogenem Netzwerk von Computern (d.h. Rechner unterschiedlicher Hersteller mit unterschiedlichen Betriebssystemen) am Fachgebiet Flugmechanik und Regelungstechnik zu ermöglichen.

5.2 Komponentenmodelle

Die komponentenbasierte Softwareentwicklung mit ihrer „pragmatischeren Objekt-Orientierung“ ([Gri99]) erfordert eigenständige Konzepte und Modelle, die die Besonderheiten und Unterschiede im Vergleich zu sonstigen Methodiken reflektieren und dadurch die Anwendbarkeit in der alltäglichen Entwicklungspraxis gewährleisten. Diese werden in der Literatur als *„Komponentenmodelle“* oder auch als *„Komponentenarchitekturen“* bezeichnet.

5.2.1 Grundlagen

Komponentenmodelle gliedern sich inhaltlich (vergleiche [Gri99]) in

1. die eigentliche Modellierung der Komponenten eines Softwaresystems sowie in
2. eine Beschreibung des dazugehörigen Gerüstes („Framework“), welches die Komponenten sinnvoll und korrekt verknüpft.

Der erste Punkt beschreibt in einer abstrakten Art und Weise, welche verallgemeinerten Rollen bzw. Aufgaben eine Komponente wahrnimmt und über welche Schnittstellen mit der Komponente interagiert werden kann. Die Beschreibung der Schnittstelle wird auch als *„Außensicht der Komponente“* bezeichnet.

Im Rollenmodell explizit enthalten sind die Freiheitsgrade oder auch Variationspunkte der Komponente. Diese legen fest, wie und auf welche Weise eine Komponente vom Entwickler angepaßt werden kann, um eine perfekte und individuelle Lösung seines Teilproblems in der zu entwickelnden Anwendung zu erarbeiten. Dieser Vorgang des *„Customizing“* ist vergleichbar der Spezialisierung einer vorgegebenen, ererbten Basisklasse, wobei eine Komponente in der Regel nicht aus einer einzigen Klasse, sondern aus einer Struktur von kooperierenden Klassen besteht. Beiden gemeinsam ist, daß die Außensicht der Klasse als auch der Komponente unverändert bleiben.

Untrennbar verbunden mit der Spezialisierung der Komponente durch einen Entwickler ist die *„Verpackung“* derselben. Hierunter ist der Auslieferungszustand der Einheit zu verstehen. Eine Komponente als Baustein einer komplexen Anwendung kann ihren vollen Nutzen nur dann entfalten, wenn gewährleistet ist, daß die Komponente in einer getesteten, garantierten Qualität vorliegt (Qualitätssicherung) und auf einfache Art und Weise mit anderen Komponenten kombiniert werden kann (Kombinierbarkeit). Bei Beachtung dieser Vorgaben sind komponentenbasierte Softwaresysteme auch wesentlich besser geeignet, sogenannte *„Altlasten“* in Neuentwicklungen zu integrieren. Altlasten sind historisch gewachsene Programme, die aus unternehmensspezifischen Gründen für ein Projekt verwendet werden müssen, und zwar so, wie sie sind. Durch eine einfache Umhüllung läßt sich das Programm in eine eigenständige, integrationsfähige Komponente verwandeln und stellt somit kein wirkliches Problem mehr dar.

Von diesem Standpunkt aus betrachtet, ermöglicht die Fähigkeit der Komponenten, zu einer komplexen Anwendung kombiniert und integriert werden zu können, die Wiederverwendung bereits bestehender Software quasi als Nebenprodukt.

Der Ansatz, für jedes Teilproblem eine möglichst perfekte Einzellösung zu erarbeiten, führt dazu, daß die Hälfte des Aufwands zur Entwicklung von Software in die Integration der Komponenten zum Gesamtsystem fließt ([Sta00]). Dieser Umstand ist zum Anlaß genommen worden, um die Komponenten herum ein Rahmenwerk zu definieren, welches dem Entwickler einer Einzelkomponente den Schritt der Integration abnehmen soll. Dieser kann dann auch automatisiert und effizienter gestaltet werden. Die prinzipielle Einbettung einer Komponente in das umgebende Gerüst zeigt Abbildung 5.1 ([Sta00]).

Jede Komponente bekommt eine *„Schutzhülle“*, einen sogenannten *Container*, der zweierlei Aufgaben erfüllt. Zum ersten verbirgt er vor der Komponente die systemspezifischen Eigenschaften, indem er Zugriffe auf das System mit Hilfe einer systemunabhängigen Schnittstelle maskiert. Desweiteren stellt er der Komponente die gewünschte Laufzeitumgebung zur Verfügung. Wie diese Laufzeitumgebung aussehen soll, muß für jede Komponente als Teil ihrer Definition spezifiziert werden.

Zum Zwecke der Vereinheitlichung wird jegliche Form von Kommunikation durch den Container hindurchgeleitet. Das betrifft Anfragen externer Klienten an die Komponente, als auch Zugriffe der Komponente auf grundlegende Dienste der Infrastruktur, wie zum Beispiel:

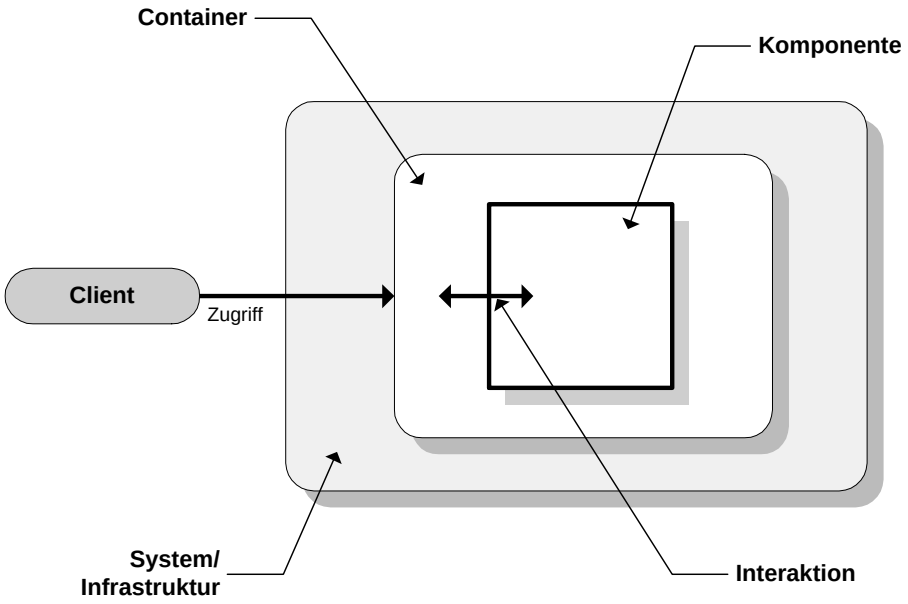


Abb. 5.1: Prinzipielle Darstellung eines Komponentenmodells

- Ereignisdienst (Event Service)
- Transaktionsdienst (Transaction Service)
- Namensdienst (Naming Service)
- Datenbankzugriffe (Persistency Service)
- Lebenszyklusmanagement (Lifecycle Service)

Es wird deutlich, daß das Rahmenwerk für die Komponenten sich in der Regel nicht nur auf die Kombination und Integration dieser beschränkt. Durch die Bereitstellung der oben beschriebenen, allgemein benötigten Funktionalitäten in einer standardisierten Art und Weise wird der Entwickler einzelner Komponenten weiter entlastet und die Mehrfachentwicklung von Software gleicher Funktionalität reduziert.

Eine Komponente wird in einer programmiersprachen-unabhängigen Weise (Hochsprache) definiert, die eine Beschreibung ihrer Schnittstelle sowie aller anderen Informationen enthält, so daß die umgebende Infrastruktur die dargestellten Aufgaben übernehmen kann. Außerdem wird meist die Definition dazu verwendet, ein *Programmier-Skelett* zu erstellen, welches dem Entwickler zugleich als Rahmen und Anleitung dient, wie er seine spezifische Einzellösung implementieren muß.

Zum gegenwärtigen Zeitpunkt können drei richtungsweisende Komponenten-Infrastrukturen angeführt werden, in denen die diskutierten, generellen Konzepte zu einer anwendbaren Technologie umgesetzt worden sind ([Sta00]):

- **Component Object Model** (COM+) von Microsoft.
- **CORBA Components** (CORC) bzw. **CORBA Component Model** (CCM) der Object Management Group (OMG)
- **Enterprise Java Beans** (EJB) von Sun Microsystems

Die drei Vertreter aktueller Komponententechnologie weisen zum Teil gravierende Unterschiede auf, sind aber im großen und ganzen durch einen hohen Grad von Übereinstimmung geprägt ([Sta00]). Aus diesem Grund ist die Diskussion eines der angeführten Beispiele ausreichend, um dem Leser ein prinzipielles Verständnis der Thematik zu ermöglichen. Zu diesem Zweck soll das *CORBA Component Model* (CCM) als das „vollständigste und flexibelste“ ([Sta00]) herangezogen und im folgenden Abschnitt näher betrachtet werden.

5.2.2 CORBA Component Model

CORBA

Die *Common Object Request Broker Architecture* (CORBA) ist eine von der *Object Management Group* (OMG) entwickelte Spezifikation, die die Zusammenarbeit von Softwaresystemen über die Grenzen von Programmiersprachen, Betriebssystemen und Netzwerken hinweg gewährleisten soll ([HC01], [Sch00b]). Zu diesem Zwecke definiert die Spezifikation eine universelle Kommunikationsplattform, mit deren Hilfe Objekte über eine Art *„Objekt-Bus“*, dem *Object Request Broker* (ORB), miteinander agieren und Daten austauschen (vergleiche Abbildung 5.2; aus [HC01]). Das Objekt, welches eine Funktionalität oder einen Dienst zur Verfügung stellt, wird im allgemeinen als Server oder im Falle von CORBA als Servant bezeichnet. Der Begriff des Client hingegen steht für das Objekt, welches die Anfrage an den Server oder Servant stellt.

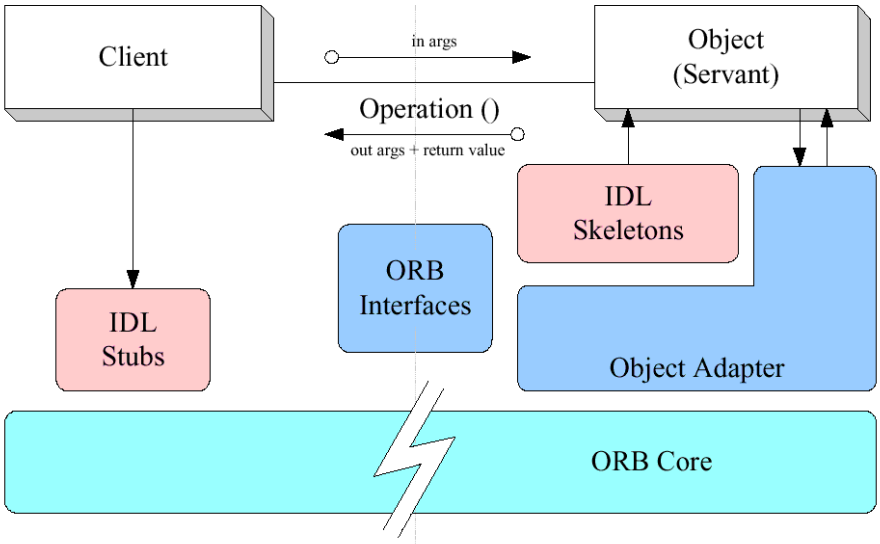


Abb. 5.2: CORBA Object Model

Der Client kann die Dienste des Servant in Anspruch nehmen ohne Kenntnis über

- Ort des Servant,
- Art der Netzwerkverbindung und der Kommunikationsprotokolle,
- betriebssystem-spezifische Belange und
- hardware-spezifische Gegebenheiten.

Die Bereitstellung der Möglichkeit, in einem heterogenen, verteilten System miteinander zu kommunizieren, stellt das „Herzstück“ der *Common Object Request Broker Architecture* dar. In Erweiterung dazu werden unter anderem sogenannte *CORBA Services* definiert. Diese repräsentieren höherwertige, wiederverwendbare Module, die dem Nutzer der Architektur elementare Funktionalitäten wie beispielsweise ein Namensdienst, ein Ereignisdienst oder Sicherheitsmechanismen über standardisierte Schnittstellen zur Verfügung stellen.

Das Komponentenmodell

Abbildung 5.3 (aus [HC01]) zeigt alle Elemente des CORBA Komponentenmodells und deren Beziehungen untereinander. Diese werden im folgenden näher erläutert.

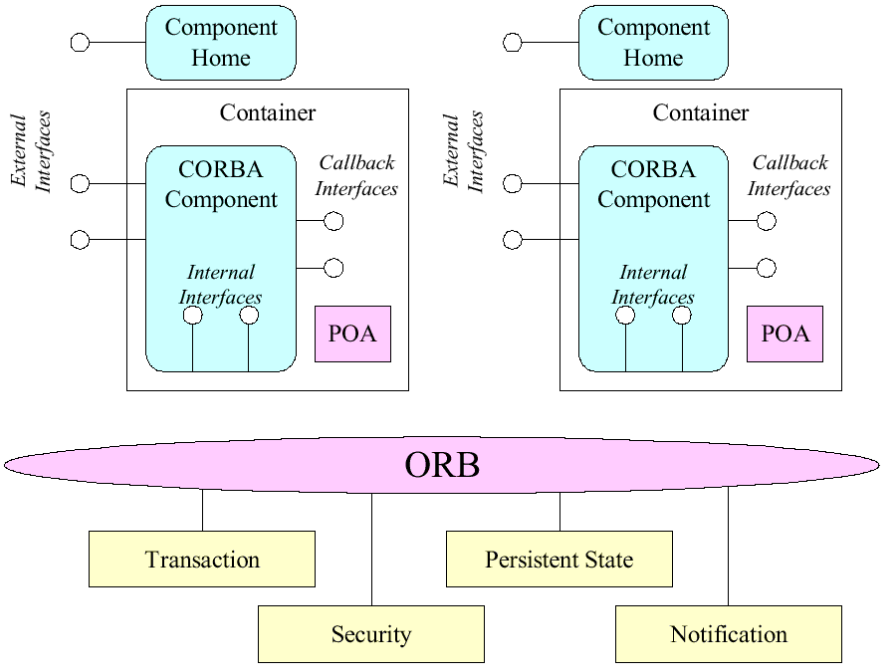


Abb. 5.3: CORBA Component Model

Zuvor sei noch darauf hingewiesen, daß eine CORBA Komponente auch gleichzeitig ein CORBA Objekt im Sinne des traditionellen *CORBA Object Models* (s.o.) darstellt. Eine Komponente wird durch eine eindeutige Referenz identifiziert, über die ein Client auf die Komponente zugreifen kann. Ist ein Client sich nicht darüber bewußt, daß es sich um eine CORBA Komponente handelt, ergibt sich für ihn kein Unterschied zu einem „normalen“ CORBA Objekt. Er kann dann jedoch nur auf die im Rahmen des traditionellen Objekt-Modells definierten und ererbten Schnittstellen des Servant zugreifen.

Externe Schnittstellen

Mit Hilfe der externen Schnittstellen („External Interfaces“ oder auch „Ports“) interagieren Klient und Komponente oder auch Komponenten untereinander. Sie stellen die eigentliche Erweiterung der Komponenten gegenüber dem traditionellen CORBA Objekt dar, indem sie der Komponente die Möglichkeit geben, ihre Funktionalität entsprechend den Ansprüchen und der Sichtweise von Benutzern differenzierter und flexibler zu gestalten.

Das CORBA Komponentenmodell definiert vier verschiedene Arten von externen

Schnittstellen:

1. *Facets*

   Ein CORBA Objekt definiert eine Menge von Operationen oder Funktionen, die ein Client auf das Objekt anwenden kann. Diese Menge wird als Schnittstelle des Objekts bezeichnet, wobei jedes Objekt nur seine eigene und eventuell vererbte Schnittstellen unterstützt. Eine Komponente hingegen kann beliebig viele Schnittstellen („facets“ oder „provided interfaces“) zur Verfügung stellen und ist dadurch in der Implementation seiner Funktionalität deutlich flexibler als ein normales CORBA Objekt. Gerade im Hinblick auf Erweiterbarkeit und Wiederverwendbarkeit ist das von großem Vorteil.

2. *Receptacles*

   Receptacles bieten die Möglichkeit Schnittstellen in eine Komponente zu importieren, d.h. die Komponente teilt mit, daß sie eine bestimmte Funktionalität nutzen, aber nicht spezifizieren und implementieren möchte. Dazu wird die Komponente mit anderen Objekten oder Komponenten verbunden, an die bestimmte Klienten-Anfragen dann delegiert werden können.

3. *Event Sources* und *Event Sinks*

   Diese Art der Schnittstelle beinhaltet die Verknüpfung der Komponente mit dem Ereignisdienst („Notification Service“ oder auch „Event Service“) der *Common Object Request Broker Architecture*. Ereignisse („Events“) stellen eine asynchrone Form der Kommunikation dar, wobei Nachrichten auch gleichzeitig an mehrere Empfänger verschickt werden können. Komponenten können über diesen „Port“ definieren, welche Ereignisse sie an welche Empfänger versenden und über welche Ereignisse anderer Komponenten sie informiert werden wollen.

4. *Attributes*

   Die Attribute einer Komponente sind eine Erweiterung der Attribute von traditionellen CORBA Objekten. Aus der Sicht von letzterem bezeichnen Attribute Eigenschaften eines Objekts, die von diesem während seiner Lebensdauer abgefragt werden können. Im Zusammenhang mit Komponenten werden Attribute auch dazu verwendet, den Status der Komponente mit Hilfe des CORBA *Component Implementation Framework* (CIF) zu konfigurieren.

Interaktion zwischen Container und Komponente

Der Container kapselt die Komponente und stellt zur Interaktion mit der Komponente zwei, von außen nicht sichtbare Arten von Schnittstellen bereit:

1. *Internal Interfaces*

   Die internen Schnittstellen werden von der Komponente genutzt, um auf die Funktionalität des Containers zuzugreifen. Diese besteht z.B. darin, Anfragen an *CORBA Services* (Transaction, Security, Persistent State und Notification Service) oder an den ORB (*Object Request Broker*) weiterzuleiten und das Ergebnis an die Komponente zurückzumelden.

2. *Callback Interfaces*

   Callbacks bezeichnen die Möglichkeit, dem Container Rückrufschnittstellen bereitzustellen, über die der Container oder der ORB (*Object Request Broker*) die Komponente über relevante Ereignisse informieren kann. Die Implementation eines Callbacks ist Aufgabe der Komponente.

Laufzeitumgebung

Der Sinn und Zweck der Kapselung durch einen Container liegt darin begründet, die Komponente vom darunterliegenden System zu entkoppeln, so daß diese kein systemspezifisches Wissen mehr benötigt. Damit fällt dem Container die Aufgabe zu, eine angemessene Laufzeitumgebung für die Komponente zur Verfügung zu stellen. In diesem Zusammenhang sind drei Aspekte von Bedeutung:

1. *Interception*

   Der Container kann ohne vollständige Kenntnis des Zustands der Komponente seine Aufgaben nicht wahrnehmen. Im Falle einer direkten Zugriffsmöglichkeit von Klienten auf die Komponente wäre diese Voraussetzung nicht mehr gegeben, so daß alle Anfragen an die Komponente vom Container abgefangen („intercepted“) werden. Durch diese Vorgehensweise ist der Container in der Lage, die aus seiner Sicht erforderlichen Maßnahmen zu ergreifen und die entsprechende Laufzeitumgebung bereitzustellen, bevor die Anfrage an die Komponente weitergeleitet wird.

2. *Persistenz*

   Für die Verwaltung der Komponenten macht es einen Unterschied, ob diese dauerhaft oder nur kurzzeitig benötigt werden. Demzufolge muß für eine Komponente festgelegt werden, welchen Persistenz-Status sie besitzen soll. Das CORBA Komponentenmodell unterscheidet vier Kategorien:

   - Service Components

     Service Komponenten sind zustandslos und an keinen festen Client gebunden.

   - Session Components

     Der Zustand einer Session Komponente ist transient, d.h. nicht dauerhaft. Für die Dauer einer Session kann dieser aber durch einen Klienten verändert werden, der der Komponente fest zugeordnet ist.

   - Process Components

     Process Komponenten sind dauerhaft, d.h. persistent, allerdings besitzt nur ein Client (*„Originalclient“*) Zugang zur Komponente.

   - Entity Components

     Diese Form der Komponente besitzt ebenfalls einen persistenten Zustand, es kann jedoch eine beliebige Anzahl von Klienten auf die Komponente zugreifen.

   Im Falle der persistenten Kategorien muß für die Komponente definiert werden, wer den Abgleich zwischen dem dauerhaften Speichermedium und der Komponente durchführt. Es wird unterschieden nach *container-verwalteten Persistenz*, wo das Laden und Speichern automatisch durch den Container erfolgt, oder nach *komponenten-verwalteten Persistenz*, wo die Datenbankzugriffe manuell durch die Komponente selbst implementiert werden.

3. *Lebenszyklus*

   Jeder Container ist für die Verwaltung des Lebenszyklus seiner Komponente verantwortlich.

   Dem Client stellt sich die Administration des Lebenszyklus in Form eines sogenannten *Home-Interface* dar. Mit Hilfe des *Home-Interface* ist der Client in der Lage, Instanzen einer Komponente aufzufinden, zu erzeugen oder auch zu zerstören. Das Auffinden des *Home-Interface* selbst geschieht über einen

in das System integrierten *Home-Finder*, vergleichbar einer Art Namensdienst für Komponenten.

Auf der Servant-Seite kommt der *Portable Object Adapter* (POA) zum Einsatz. Dieser ist schon im Rahmen des traditionellen CORBA Objekt-Modells für die Aktivierung und Deaktivierung von Servants zuständig. Außerdem sorgt er für eine eindeutige Zuordnung der von den Klienten benutzten CORBA Objektreferenzen zur entsprechenden Servant-Implementation. Die Art und Weise der Aktivierung bzw. Deaktivierung kann aus folgenden Verfahrensweisen ausgewählt werden:

- Method

  Die Aktivierung einer Komponente erfolgt direkt vor dem Methodenaufruf. Nach Abarbeiten der Methode wird die Komponente wieder deaktiviert.

- Session

  Hier erfolgt die Aktivierung beim ersten Aufruf durch den Client. Die Deaktivierung ist an die Lebensdauer des Klienten gebunden.

- Component

  Mit dem ersten Aufruf wird die Komponente in einen aktivierten Zustand versetzt und erst wieder auf explizite Anordnung deaktiviert.

- Container

  Eine Aktivierung wird durch die erste Anfrage initiiert und der Container entscheidet, wann die Komponente deaktiviert wird.

Softwareerstellung und -verteilung

Der erste Schritt in der Entwicklung einer CORBA Komponente (siehe Abbildung 5.4; aus [HC01]) ist gleich dem eines traditionellen CORBA Objekts. Mit Hilfe der Hochsprache IDL (*Interface Definition Language*) wird die Schnittstelle eines Objekts festgelegt, wobei der Umfang der IDL aufgrund neuer Aspekte durch das Komponentenmodell erweitert worden ist. Ein IDL-Compiler erzeugt aus der Beschreibung die für ein CORBA Objekt üblichen *Client Stub* und *Server Skeleton*. Ersteres bildet die Basis für den Zugriff eines Client, letzteres stellt das Skelett dar, mit dessen Hilfe der Servant implementiert wird.

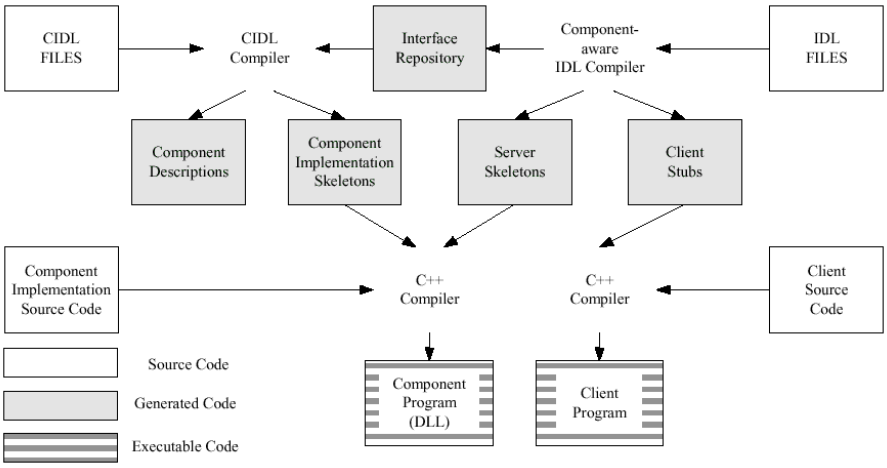


Abb. 5.4: Komponentenimplementation

Da die *Interface Definition Language* keine Möglichkeit bietet, Implementations- und Persistenzaspekte zu beschreiben, wird eine weitere Hochsprache, die *Component Implementation Definition Language* (CIDL) eingeführt. Mit ihr kann beispielsweise festgelegt werden, welche Persistenz-Policy verwendet werden soll. Ein CIDL-Compiler verwendet diese Information, um eine Beschreibung der Komponente und ein weiteres Skeleton zu generieren, in diesem Fall das *Component Implementation Skeleton*.

Als Abschluß der Implementation steuert der Entwickler Programm-Code bei, der das spezifische Verhalten der Komponente umsetzt. Geschieht das z.B. in der Programmiersprache *C++*, dann werden durch einen entsprechenden *C++-Compiler* der *Component Implementation Source Code*, das *Component Implementation Skeleton* und das *Server Skeleton* zu einem ausführbaren Programm-Code zusammengesetzt. Ein Client, auf der anderen Seite, benötigt nur den *Client Stub* und sein *Client Source Code*, um daraus ein Client Programm zu erzeugen, welches auf die Komponente zugreifen kann.

In diesem Zustand kann die Komponente jedoch nicht ausgeliefert werden. Um in einer komponenten-basierten Anwendung zum Einsatz zu kommen, wird die Komponentenimplementation mit sogenannten *Metainformationen* versehen und alles zusammen in einem Zip-Archiv, sogenannte *CAR-Files*, verpackt. Die Metainformationen beschreiben vereinfacht formuliert, wie die Komponenten-Software verwendet werden kann und wie sie auf der Zielplattform installiert und konfiguriert werden muß. Dabei soll der Anwender durch ebenfalls spezifizierte Werkzeuge unterstützt werden.

5.2.3 Bewertung

Komponentenmodelle stellen eine Ergänzung der bisherigen objekt-orientierten Softwareentwicklung dar. Sie werden mit ihrer Softwaresicht der Anwendungsebene (Nutzung, Wartung, Auslieferung und Update von Software) wesentlich eher gerecht als bisherige Konzepte ([Gri99]). Dadurch wird der Entwicklungsprozeß den Gegebenheiten der Realität angepaßt und komplettiert.

Für die Unternehmen der Softwarebranche ist die komponenten-basierte Softwaresicht ein Mittel zur Industrialisierung des Entwicklungsprozesses, indem es eine klare Aufgabenverteilung zwischen Komponenten- und Anwendungsentwickler vorgibt und Potential zur Rationalisierung verspricht. Auf der anderen Seite birgt es aber auch die Gefahr, einen Trend zur „Wegwerf"-Software einzuleiten ([Gri99]). Wenn man „mal eben" ein paar Komponenten zu einer Anwendung zusammenstellt, die heutzutage eine durchschnittliche Lebenserwartung von einem halben Jahr besitzt, dann ist eine Wiederverwendung dieser aus einer betriebswirtschaftlichen Sicht heraus nicht unbedingt angeraten.

Im Mittelpunkt der Komponentensicht steht die Integrationsproblematik ([Gri99]). Die Frage, wie viele verschiedenen Bausteine unterschiedlichster Hersteller beispielsweise in einem Unternehmen effizient und sinnvoll zusammenwirken können, charakterisiert das eigentliche zentrale Problem heutiger heterogener EDV-Landschaften. Mit der starken Betonung der Eigenständigkeit und Unabhängigkeit der Einzelbausteine scheinen die Komponentenmodelle der erfolgreichste Kandidat für die Lösung dieser Problematik zu sein.

Die Tatsache, daß die Programmierung verteilter Systeme mittlerweile zur Tagesordnung gehört, wird in diesem Zusammenhang die Probleme nicht reduzieren, sondern eher vergrößern. Jedoch ist die Komponententechnologie aufgrund ihrer Verwandschaft mit den verteilten, objekt-orientierten Systemen auch hier durchaus einsetzbar. Die starke, gegenseitige Abgrenzung und Entkopplung der einzelnen kommunizierenden Softwareteile in einem verteilten Systems war einer der Ursprünge, aus denen heraus sich die Idee der komponenten-basierten Softwareentwicklung entwickelt hat.

Nachteilig für eine Verwendung von Komponententechnologien sind die heutzutage noch hohen Ressourcenanforderungen bestehender Implementationen und das unvorhersagbare Laufzeitverhalten ([Sta00], [WLS00]). Durch die Bereitstellung infrastruktureller Dienste und die maximal mögliche Entlastung des Komponenten-Entwicklers ist die Benutzung einer Komponententechnologie mit einem nicht immer

akzeptablen „Overhead“ hinsichtlich Zeit und Ressourcen-Verbrauch verbunden, beispielsweise für Echtzeit- und High-Performance-Anwendungen. Als Konsequenz daraus gibt es Bestrebungen, das *CORBA Component Model* und die zugrundeliegende *Common Object Request Broker Architecture* für einen Einsatz in Echtzeitsystemen zu optimieren (*Realtime CORBA*), so daß die allgemeinen Dienste in einer garantierten Qualität zur Verfügung gestellt werden können (*Quality of Service*).

Die Bereitstellung einer integrierten Lösung ist auch aus Entwicklersicht nicht zu unterschätzen. Dem Fortschritt verteilter Systeme können die Entwicklungsumgebungen in der Regel nicht Schritt halten, so daß man sagen muß, daß das Testen und Debuggen verteilter, komponenten-basierter Softwaresysteme immer noch mühsam und meist manuell erfolgen muß ([Sta00]). Administration und Konfiguration verteilter Systeme wird ebenfalls häufig von Herstellern vernachlässigt, obwohl es eigentlich in der Theorie als ein wichtiger Bestandteil des Entwicklungsprozesses adressiert worden ist.

Eine weitere Diskrepanz zur Theorie wird daran erkennbar, daß die Implementationen verschiedener Hersteller trotz standardisierter Spezifikationen keine befriedigende Interoperabilität gewährleisten ([Sta00]), so daß häufig Unternehmen auf „Monokulturen“ einzelner Firmen setzen müssen.

Diese Problematik wird auch deutlich bei der Betrachtung der unterstützen Programmiersprachen und Plattformen. *Component Object Model* (COM+) von *Microsoft* ermöglicht zwar die Verwendung verschiedener Programmiersprachen, ist aber auf den Einsatz mit der Plattform *Windows* beschränkt. Das Konkurrenzprodukt von *Sun Microsystems*, *Enterprise Java Beans* (EJB), zeigt das umgekehrte Verhalten, es läuft auf nahezu allen Betriebssystemen, unterstützt aber nur die Programmiersprache *Java*. Einzig das *CORBA Component Model* (CCM) ist weder auf eine Programmiersprache, noch auf ein Betriebssystem festgelegt. Der Haken ist jedoch, daß die Spezifikation des *CORBA Component Model's* erst vor kurzem fertiggestellt worden ist, so daß die Anzahl verfügbarer Implementationen noch gering ist und man davon ausgehen kann, daß die vorhandenen Umsetzungen noch eine Zeit der Reifung benötigen, in der zu erwartende „Kinderkrankheiten“ ausgemerzt werden können ([Sta00], [WLS00]).

Unabhängig davon, daß sowohl auf theoretischer als auch auf Implementationsseite eine Phase der Konsolidierung erfolgen muß, sind die Komponententechnologien kein aktueller Modetrend, der bald wieder verschwinden wird. Ihr Einsatz ist schon jetzt in vielen Projekten möglich und wird in Zukunft noch zunehmen ([Sta00]).

5.3 High Level Architecture

Die Spezifikation der *High Level Architecture* (HLA) ist vom amerikanischem *Department of Defense* (DoD) entwickelt worden und entstammt der Erkenntnis, daß die Einsatz- und Kombinationsmöglichkeiten von neuen und bestehenden Simulatoren nicht vorhergesehen werden können. Deswegen wurde der Ansatz versucht, Simulationen aus einzelnen Bausteinen zusammensetzen zu können. Das Ergebnis dieser Überlegungen ist die *High Level Architecture*; sie stellt eine Infrastruktur bereit, in der die Interaktion verschiedenster Modelle und Simulationen in einem verteilten System ermöglicht wird.

Die *High Level Architecture* integriert die Vorgängertechnologien wie *Distributed Interactive Simulation* (DIS) und *Aggregate Level Simulation Protocol* (ALSP) und ist mittlerweile der Standard für alle Simulationen des *Department of Defense*. Zusätzlich ist sie auch als Standard durch das *Institute of Electrical and Electronic Engineers* (IEEE) aufgenommen.

5.3.1 Allgemeines

Der *High Level Architecture* kann im weitesten Sinne als ein Komponentenmodell aufgefaßt werden, da sie zwei wesentliche Merkmale der Komponentensicht aufweist (vergleiche Abbildung 5.5; aus [Dah98]):

- Definition von modularen Komponenten mit eindeutig definierter Funktionalität und Schnittstelle.
- Spezifische Simulationsbelange werden von einer allgemeingültigen Infrastruktur getrennt, welche grundlegende Dienste zur Verfügung stellt.

Im Gegensatz zu den vorher vorgestellten, allgemeinenen Komponentenmodellen ist die *High Level Architecture* jedoch auf den Einsatz in verteilten Simulationen ausgerichtet, wobei sie grundsätzlich für den gesamten Anwendungsbereich der Simulation (Training, Forschung, Analyse, etc.) konzipiert worden ist. Bestandteile einer HLA-Simulation können nicht nur Simulationsprogramme, sondern auch sogenannte „Live Players“ (Informationen über reale Systeme) oder der Mensch sein.

In der Begrifflichkeit der *High Level Architecture* wird das verteilte Simulationsmodell als *Federation* und die einzelnen Komponenten als *Federates* bezeichnet. Die

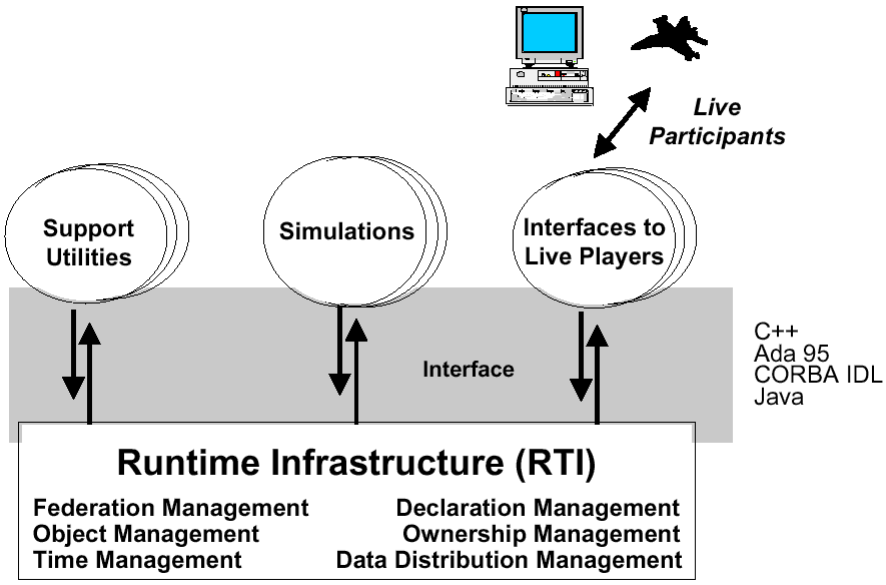


Abb. 5.5: Übersicht High Level Architecture

Federates dürfen beliebig komplex sein, beispielsweise können sie ein gesamtes Flugzeug in einem Simulationsverbund oder auch nur eine einzelne Anzeige in einem Flugzeugcockpit darstellen. Die *Federation* und ihre *Federates* werden durch das unterstützende Rahmenwerk, die *Runtime Infrastructure* (RTI), und durch ein die Federation beschreibendes Objektmodell (*Federation Object Model* (FOM)) komplettiert.

Die Runtime Infrastructure ermöglicht die Kommunikation zwischen den Federates, die untereinander keinerlei weitere Verbindung aufweisen. Kommunikation findet einzig und allein über die RTI statt.

Das Objektmodell beschreibt im wesentlichen

- die Komponenten in der Simulation
- und deren Beziehungen untereinander.

Es wird von der *Runtime Infrastructure* zur Ausführung der Simulation (*Federation Execution*) benötigt.

Das Objektmodell und die Schnittstelle zur RTI werden im folgenden eingehender beschrieben.

5.3.2 Objektmodell

Die Objektmodelle in der HLA (vergleiche [SK99]) geben Aufschluß über

- die Menge von Objekten, die in der Simulation die reale Welt repräsentieren,
- die Eigenschaften, Beziehungen und Interaktionen dieser Objekte und
- die Abbildungsgenauigkeit („level of detail“) der Objekte hinsichtlich des repräsentierten, realen Systems.

Als Anleitung zur Erarbeitung der Objektmodelle und gleichzeitig als deren Dokumentation dient das *Object Model Template* (OMT). Es strukturiert die von der *Runtime Infrastructure* (RTI) zur Ausführung benötigten Informationen wie folgt:

Object Class Structure Table

In der HLA sind *Objektklassen* persistente Objekte, die sich nur durch ihre Attribute auszeichnen. Auf diese Attribute können die *Federates* einer *Federation* schreibend und lesend während einer *Federation Execution* zugreifen. Die *Object Class Structure Table* ist eine statische Beschreibung aller in einer Federation bekannten Objektklassen.

Interaction Class Structure Table

Sogenannte Interaktionsklassen (*Interaction Classes*) beschreiben die dynamischen Zusammenhänge, die Interaktionen, zwischen den *Federates* in einer *Federation*. Nur durch das Versenden von Instanzen von Interaktionsklassen können *Federates* miteinander interagieren. Diese Klassen enthalten die für eine Interaktion benötigte Information in Form von Attributen, welche immer in ihrer Gesamtheit ausgetauscht werden müssen. Die Übergabe einzelner Attribute oder Parameter einer Interaktionsklasse ist nicht möglich.

Instanzen einer Interaktionsklasse sind transient, d.h. sie existieren nur für die Dauer der Interaktion.

Die *Interaction Class Structure Table* ist eine statische Auflistung der möglichen Interaktionsformen zwischen den Objekten einer *Federation* und der dabei betroffenen Attribute.

Attribute/Parameter Table

Diese Tabelle enthält Informationen zu Attributen und Parametern der verwendeten Klassen in einer *Federation*, beispielsweise welchen Datentyp besitzt ein Attribut (*float*, *double*, *short*, etc.). Diese Information findet jedoch nicht Eingang in die Infrastruktur, sondern ist eine reine Dokumentation. Jeder Entwickler eines *Federates* muß folglich vor Laufzeit der Simulation sicherstellen, daß die dort beschriebenen Datentypen mit den von ihm erwarteten übereinstimmen.

Complex Data Type Table

Die von der HLA definierten Basistypen (*float*, *double*, *short*, etc.) können von den Entwicklern in Form von *Complex Data Types* erweitert werden (nutzer-definierte Typen). Darunter fallen auch beispielsweise Aufzählungstypen. Mit Hilfe der *Complex Data Type Table* werden solche Erweiterungen allen *Federates* einer *Federation* bekannt gemacht.

Routing Space Table

Der *Routing Space Table* spezifiziert und koordiniert den Datenaustausch zwischen den Mitgliedern einer *Federation* (vergleiche auch Abschnitt 5.3.3).

Simulation Object Model (SOM)

Das *Simulation Object Model* (SOM) beschreibt die Eigenschaften eines *Federates*. Darunter fällt im wesentlichen, welche Information, in Form von Objekten und Attributen, kann es der *Federation* zur Verfügung stellen, und welche Aktionen beabsichtigt es auszuführen.

Federation Object Model (FOM)

Das *Federation Object Model* (FOM) kann als ein „Vertrag“ aller *Federates* untereinander verstanden werden. Es beschreibt hauptsächlich die Summe aller Objekt- und Interaktionsklassen (statische und dynamische Aspekte), die von allen *Federates* der *Federation* geteilt werden. Das *Federation Object Model* wird der *Runtime Infrastructure* in Form einer Datei (*Federation Execution Data* (FED)) vor der Laufzeit zugänglich gemacht.

5.3.3 Interface Spezifikation

Jedes *Federate* besitzt nur **eine Schnittstelle**, nämlich die zur *Runtime Infrastructure* (s.o.). Abbildung 5.6 (aus [SK99]) zeigt das Prinzip der Kommunikation zwischen *Federate* und RTI.

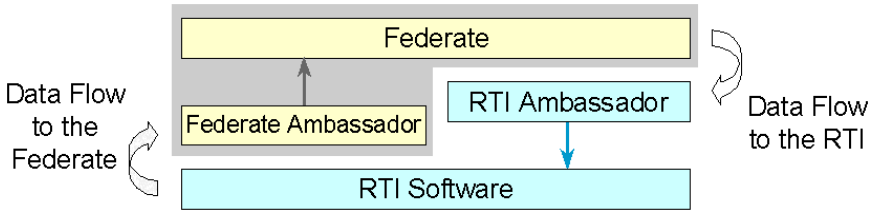


Abb. 5.6: Kommunikation mit Hilfe von *Ambassador* Objekten

Der Datenaustausch zwischen Federates und RTI erfolgt mit Hilfe von Objekten, den sogenannten *Ambassadors*. Dabei kommt der *RTI Ambassador* zum Einsatz, wenn das Federate die RTI ansprechen möchte, und umgekehrt der *Federate Ambassador*, wenn die RTI dem Federate etwas mitzuteilen hat.

Der *RTI Ambassador* wird von der Infrastruktur bereitgestellt, wohingegen der *Federate Ambassador* vom Entwickler des *Federates* erstellt werden muß. Die Schnittstelle zur *Runtime Infrastructure* wird für folgende Programmiersprachen angeboten:

- *C++*
- *ADA95*
- *Java*

Neben der Bereitstellung der reinen Kommunikationsstruktur gehört auch das Angebot grundlegender Dienste zu den Aufgaben der *Runtime Infrastructure*; diese lassen sich in sechs Gruppen unterteilen:

Federation Management

Das *Federation Management* dient der Koordination der Ausführung von *Federations*. Koordination bedeutet in diesem Zusammenhang hauptsächlich Lebenszyklus-Management der *Federation* (Erzeugung, Zerstörung, Speichern, Wiederherstellen). In den Verantwortungsbereich des *Federation Management* fällt aber auch die Verwaltung der An- und Abmeldungen einzelner Federates, die auf diese Weise dokumentieren, ob sie an einer Simulation teilnehmen oder ob sie diese verlassen wollen.

Declaration Management

Über das *Declaration Management* geben *Federates* bekannt, welche Datentypen (Objekt- und Interaktionsklassen) sie empfangen oder senden möchten. Ersteres bezeichnet die RTI mit dem Begriff *„subscribe“*, letzteres mit *„publish“*. Beide Möglichkeiten beziehen sich auf Klassen und Attribute, nicht auf spezifische Instanzen. Das bedeutet, daß ein *Federate* Informationen von allen Instanzen eines Typs empfängt, wenn er diesen Typ der RTI zum Lesen angemeldet hat.

Das *Declaration Management* sorgt auch dafür, daß Empfänger immer mit aktuellen Werten versehen werden, indem sie den produzierenden *Federates* Rückmeldung darüber gibt, wann diese einen *Update* ihrer „veröffentlichten“ Objekte durchführen sollen.

Object Management

Das *Object Management* verwaltet Objekt-Instanzen innerhalb einer *Federation*. Mit seiner Hilfe können Instanzen von Objekt- und Interaktionsklassen erzeugt, verändert und zerstört werden. Dadurch werden Objekte und Attribute den anderen Teilnehmern der Simulation bekannt gemacht. Erst wenn ein neues Objekt über die *Object Management*-Dienste registriert (*„register“*) worden ist, ist es für eine andere Komponente sichtbar (*„discover“*), und es kann zu einen Austausch zweier Federates kommen (senden und empfangen bzw. *„update“* und *„reflect“*).

Die analogen Funktionalitäten werden in bezug auf Interaktionsklassen mit *„send“* und *„receive“* bezeichnet. Die Erzeugung und Zerstörung der Instanzen von Interaktionsklassen geschieht implizit zu Beginn und zum Ende der Transaktion.

Über das *Object Management* wird auch die *Transport Policy* und die *Receive Order* festgelegt. Ersteres charakterisiert die Bedeutung der übertragenen Information. Muß sie auf jeden Fall beim Empfänger ankommen, so wird ein sicherer Transport-Mechanismus (*„reliable“*) verwendet. Kann dagegen ein Datenverlust, beispielsweise im Hinblick auf einen Performance-Gewinn, in Kauf genommen werden, dann geschieht die Übermittlung der Daten nach bestmöglichen Verhältnis von Nutzen und Aufwand (*„best effort“*).

Die *Receive Order* legt fest, ob Informationen in der Reihenfolge, in der sie bei der RTI eingehen, weitergegeben werden sollen (*„receive order“*), oder ob sie nach ihren Zeitstempeln geordnet ausgeliefert werden sollen (*„time stamp order“*).

Data Distribution Management

Das *Data Distribution Management* hat die Aufgabe, den Datenaustausch zwischen den Federates zu optimieren, um beispielsweise die damit verbundene Netzwerkbelastung zu reduzieren. Die HLA führt dazu die Idee der *Routing Spaces* ein (vergleiche auch Abschnitt 5.3.2), mit Hilfe derer ein *Federate* festlegen kann, in welchem Wertebereich eine Information für ihn interessant ist, so daß diese auch nur in einem entsprechenden Fall übertragen werden muß. Ähnliches gilt für das Versenden von Daten. Ein *Federate* kann im Vorhinein mitteilen, in welchem Wertebereich sich die Informationen, die er produziert, bewegen werden.

Ein anschauliches Beispiel wäre die Verwendung eines *Traffic Collision Avoidance System* (TCAS). Dieses errechnet im Falle eines möglichen Zusammenstosses von Flugzeugen eine Flugroute, mit deren Hilfe ein Zusammenstoß vermieden werden kann. Für die Berechnung der Ausweichroute benötigt das System Informationen über andere Flugzeuge, jedoch nur über die in unmittelbarer Umgebung. Durch die Spezifikation einer Region um die aktuelle Position des Flugzeugs herum, kann das *Data Distribution Management* dafür Sorge tragen, daß nur Daten über Fremd-Flugzeuge in unmittelbarer Nähe an das Kollisionsvermeidungs-System übermittelt werden.

Die Verwendung des *Data Distribution Management* ist optional in Bezug auf die Ausführung einer *Federation*.

Time Management

Mit Hilfe des *Time Managements* wird definiert, wie der Zeitfortschritt in der Simulation erfolgen soll. Dazu teilen die *Federates* dem *Time Management* mit, ob sie aktiv (*„time-regulating“*) oder passiv (*„time-constrained“*) das Voranschreiten der Zeit gestalten wollen.

Diese Information und das Wissen über die Art und Weise, wie die *Federates* ihre Nachrichten zugestellt bekommen möchten (s.o.), geben der *Runtime Infrastructure* die Möglichkeit, den Zeitfortschritt für die gesamte *Federation* zu koordinieren und mit der Nachrichtenübermittlung in Einklang zu bringen.

Zeitinformationen in der HLA haben immer nur aus der Sicht des betreffenden *Federates* Gültigkeit und nicht absolut für alle Teilnehmer einer *Federation*. Dadurch kann jedem *Federate* das lokale Zeitmanagement überlassen werden, ohne daß es für

andere von außen sichtbar wird. Die HLA übernimmt nur die globale Koordination und ermöglicht so jedem *Federate* den für ihn passenden Algorithmus (ereignis-orientiert, takt-orientiert, echtzeit-orientiert, etc.).

Die RTI sorgt nicht für eine Angleichung an die physikalische, reale Zeit, wie sie vom Menschen verstanden wird. Dies kann nur durch einen *Federate* erfolgen, der den Zeitfortschritt aktiv mitgestaltet und sich dabei beispielsweise an einer Rechner-Uhr orientiert.

Ownership Management

Mit Hilfe des *Ownership Managements* können die Eigentumsrechte von Objekten und Attributen (nicht aber Interaktionen) festgelegt und verändert werden. Die Bedeutung der Eigentumsrechte liegt darin, daß nur Attribute, die einem *Federate* gehören, auch von diesem geändert werden dürfen.

Eigentumsrechte können nur während der Ausführung einer Simulation (*Federation Execution*) angefordert und abgegeben werden, wobei das *Ownership Management* dann dafür sorgt, daß alle davon betroffenen *Federates* über die Änderung benachrichtigt werden.

Es besteht auch die Möglichkeit, die Eigentumsrechte von Attributen eines Objekts auf mehrere *Federates* zu verteilen.

5.3.4 Entwicklungsprozeß

Die *High Level Architecture* beschreibt in Anlehnung an den Lebenszyklus eines Simulationsmodells einen fünf-stufigen Entwicklungsprozeß eines Federates (siehe Abbildung 5.7; aus [Dah98]).

Dieser Lebenszyklus wird zunehmend durch die Verwendung von Tools unterstützt, die den Entwickler weiter entlasten und den Gesamtvorgang stärker automatisieren sollen. Als Beispiel wäre zu nennen das *Object Model Development Tool* des *Defense Modeling And Simulation Office* (DMSO) zur Erstellung von HLA Objektmodellen.

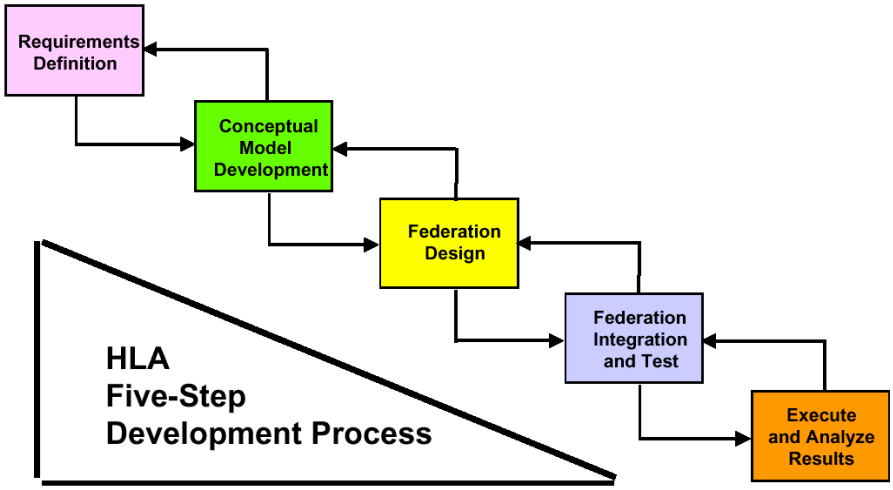


Abb. 5.7: Entwicklungsprozeß eines *Federates*

5.3.5 Bewertung

Die *High Level Architecture* stellt einen großen Funktionsumfang zur Konfiguration und Durchführung einer verteilten Simulation zur Verfügung ([Eng01]).

Dabei wird sie den Ansprüchen nach Transparenz, Skalierbarkeit und Unabhängigkeit vom Simulationsmodell gerecht. Von besonderer Bedeutung im Zusammenhang mit interaktiver Simulation (der Mensch in Interaktion mit der durch die Simulation bereitgestellten virtuellen Umgebung) sind auch die Möglichkeiten zur Strukturierung des Simulationsmodells und die Gestaltung des Zeitfortschritts (Synchronisation) in der Simulation. Beide Aspekte sind wesentliche Kriterien dafür, wie gut die Abbildungsgenauigkeit hinsichtlich des realen Systems ist und inwiefern der Mensch sich in die virtuelle Umgebung hineinversetzen kann.

Der Einsatz der *High Level Architecture* in Simulationsvorhaben wird für mehrere Programmiersprachen (*C++*, *ADA95* und *Java*) und die gebräuchlisten Betriebssysteme ([Dah98]) ermöglicht. Ein nicht zu unterschätzender Nachteil für Forschungs- und Entwicklungssimulatoren ist aber dadurch gegeben, daß Implementationen der Infrastruktur nur in Binärform und nicht als Quellcode verfügbar sind.

Schwerwiegender als die Einschränkung in den Arbeitsmitteln wiegt jedoch die Tatsache, daß die Konzeption der *High Level Architecture* Ansprüche hinsichtlich dynamischer Konfiguration und Flexibilität in der Ausführung nur in geringerem Maße berücksichtigt hat. Die Forderung nach Kombinierbarkeit und Wiederverwend-

barkeit von Simulator-Komponenten ist zwar als das Hauptziel formuliert worden, darüber ist jedoch die Benutzerfreundlichkeit und die Möglichkeit zur schnellen, dynamischen Rekonfiguration in den Hintergrund gedrängt worden ([Eng01]). Das spiegelt sich auch darin, daß HLA häufig als ein Rahmenwerk für große, monolithische Simulationen angesehen und dementsprechend eingesetzt wird ([CH99]). Ein Forschungssimulator wird aber nicht einmal aufgebaut und dann nur noch so verwendet, wie er ist, sondern er ist einem dauernden Wandel unterworfen (soft- und hardwareseitig), der sich aufgrund auftretender Probleme oder neuer Anforderungen durch Forschungsvorhaben fast zwangsläufig ergibt.

Für den Entwickler kommt erschwerend hinzu, daß die HLA-Spezifikation sehr „technisch“ gehalten ist ([BK00]). Sie erfordert viel Wissen und ist nicht einfach zu verwenden, sobald man sich auf die Programmiersprachen-Ebene herunter begeben hat ([SK99]). Viele kommen deswegen zur Aussage, daß die manuelle Entwicklung von HLA Objektmodellen kompliziert, anspruchsvoll und damit fehleranfällig ist ([GGBR98], [Nau99]).

Insbesondere der Vergleich mit existierenden Simulationstools, die den Entwickler vor dieser Art „low-level programming“ bewahren, führt zu dem Ansatz die Programmierung von HLA Objekten durch eine abstraktere, höherwertige Entwickler-Schnittstelle zu ermöglichen („middleware layer approach“, vergleiche [BK00]).

Der *High Level Architecture* fehlt nach diesen Aussagen eine Art integrierte Entwicklungsumgebung (Integrated Development Environment (IDE)), die es ermöglicht, Simulationskomponenten auf eine strukturierte, kontrollierte und auch effiziente Art und Weise zu erzeugen. Gerade im Hinblick auf die Iterationsmöglichkeiten im Lebenszyklus eines Simulationsmodells (vergleiche Kapitel 2.1.3) ist eine derartige Unterstützung des Entwicklers auf jeden Fall angeraten.

Als Konsequenz zur fehlenden Automatisierung und Produktivität in der Entwicklung und Ausführung von HLA Simulationen werden vermehrt Tools bzw. infrastrukturelle Erweiterungen ([Cox98], [GGGH98]) von dritter Seite angeboten, die sich der Problematik annehmen. Hier läßt sich ein ähnlicher Trend wie bei den Komponentenmodellen erkennen (siehe Abschnitt 5.2), wo der Entwicklungsprozeß nicht mit der Implementation der Komponente beendet ist, sondern die Bereiche Integration, Installation, Nutzung und Wartung miteinbezieht.

Ein weiterer Nachteil der *High Level Architecture* besteht darin, daß der Entwickler einer einzelnen Komponente noch relativ viel Wissen über die Infrastruktur (RTI) und die anderen Komponenten in einer *Federation* haben muß bzw. Aufgaben, die

eigentlich der Infrastruktur zuzurechnen wären, übernehmen muß ([Eng01], [BK00], [CP98]).

So werden allgemein Objekte zur Laufzeit nur über einen Namen identifiziert. Die *Runtime Infrastructure* hat kein Wissen darüber, was sich hinter dem Objekt verbirgt, und kann demzufolge auch keine Überprüfung vornehmen, ob die zwei Seiten einer Kommunikationsverbindung die gleiche Vorstellung des durch den Namen beschriebenen Objekts besitzen. Die Verantwortung wird den Entwicklern von *Federates* überlassen, die sich während der Entwicklungsphase, also vor der Laufzeit, untereinander absprechen müssen. Gerade im Hinblick auf maschinen- und compilerspezifische Besonderheiten der Datenrepräsentation können sich dadurch schwer detektierbare Fehler in die Anwendung einschleichen.

Ein Nebeneffekt davon ist, daß ein übergeordneter Beobachter der *Federation* sich keinen wirklichen Überblick über den Zustand der Simulation verschaffen kann. Die einzige, zentrale Anlaufstelle, die Informationen zur Verfügung stellen kann, ist die RTI, welche nur die Namen der Objekte in der Simulation kennt, nicht aber deren Status und Bedeutung.

Schließlich lassen sich im Zusammenhang mit der *High Level Architecture* noch einige technische Probleme anführen. In der HLA wird versucht, den Datenaustausch über das Netzwerk so weit wie möglich zu optimieren. Die dadurch erreichten geringen Ansprechzeiten bei der Informationsvermittlung werden mit einem „Overhead“ hinsichtlich Speicherbedarf und Synchronisation erkauft. Denn für jedes Objekt, auf das zugegriffen werden soll, wird auf dem Rechner des Empfängers eine lokale Kopie des eigentlich referenzierten Objekts eingerichtet ([BK00]).

Werden auf den verschiedenen, an der Simulation beteiligten Computern Implementationen der *Runtime Infrastructure* von anderen Herstellern als dem *Department of Defense* eingesetzt, so ist die Interoperabilität dieser nicht spezifiziert und deswegen auch nicht gewährleistet ([BK00]). Als Lösung wird beispielsweise angedacht, das im Zusammenhang mit CORBA definierte *Internet-Inter-ORB-Protocol* (IIOP) verbindlich für den Austausch zwischen den RTI's einer verteilten Simulation vorzuschreiben.

Problematisch gestaltet sich auch durchaus die Anpassung bestehender Simulatoren, die sich beispielsweise der Vorgängertechnologie *Distributed Interactive Simulation* (DIS) bedienen. Trotz der Zielsetzung, den Übergang von DIS auf HLA so einfach wie möglich zu gestalten, enthüllt die Umsetzung unvorhergesehene Komplexitäten und ist mit relativ hohen Aufwand verbunden. Selbst eine erfolgreiche Transition

auf HLA gewährleistet dann nicht unbedingt, daß eine Simulationskomponente unter DIS und HLA das gleiche Verhalten aufweist ([CP98]).

5.4 Anwendungsbeispiele

In Ergänzung zur vorherigen theoretischen Diskussion über Ideen und Konzepte zur Verwaltung einer verteilten Simulation sollen an dieser Stelle in aller Kürze einige Anwendungsbeispiele aus der Praxis exemplarisch aufgeführt werden.

5.4.1 Allgemein

Das Management einer verteilten Anwendung ist keine Problemstellung, die sich auf den Bereich verteilte Simulation beschränkt; dies ist vielmehr der Spezialfall. Strukturen, die sich der Verwaltung einer verteilten Applikation von einem allgemeineren Standpunkt aus annehmen, setzen in der Regel weniger Wissen über die zu organisierenden Prozesse voraus, können deswegen aber auch der Anwendung nur in geringerem Maße grundlegende Dienste zur Verfügung stellen.

- **Jini Management Desktop**

  Der *Jini Management Desktop* ist eine Architektur zur Bereitstellung beliebiger Dienste in einem Netzwerk, die an der Technischen Universität Darmstadt, Fachgebiet Informatik, entwickelt worden ist ([AH01]). Demonstrationsszenario ist in diesem Fall die Administration der Ressourcen eines Netzwerkes, beispielsweise die Überwachung aller Drucker in einem Unternehmen, so daß diese stets betriebsbereit sind und nicht übermäßig belastet werden.

  Alle Dienste werden durch eigenständige, spezifische Prozesse bereitgestellt, deren Aufgabe in der Verwaltung einer oder mehrerer Ressourcen besteht. Die Prozesse können sich über eine zugrundeliegende Kommunikationsstruktur untereinander austauschen und werden über eine zentrale Ebene, die alle Informationen über den Zustand des Systems verwaltet, koordiniert.

  Die Betonung der eigenständigen Applikation als Diensteanbieter, die über eine Infrastruktur mit anderen Einheiten kommuniziert, entspricht der in Abschnitt 5.2 vorgestellten Komponentensicht. Ein vollständiges Komponentenmodell, welches sich auch mit Aspekten wie Integration, Installation, Nutzung und Laufzeitumgebung der Komponente auseinandersetzt, wird in diesem Zusammenhang jedoch nicht vorgestellt.

- **Management of Distributed Applications and Systems (MANDAS)**

  MANDAS ist eine Entwicklung der University of Western Ontario, Kanada ([BBER+97]) mit dem Ziel, die Ausführung einer verteilten Anwendung, repräsentiert durch eine Anzahl von Prozessen und den von ihnen benötigten Dateien, zu unterstützen.

  Das Rahmenwerk baut auf der vorhandenen Netzwerkinfrastruktur und dem Betriebssystem auf und unterteilt seine grundlegenden Dienste in die Bereiche Überwachung, Kontrolle und Konfiguration. Die eigentliche Umsetzung geschieht durch die *Management Applications*, die beispielsweise die Performance eines Systems überwachen. MANDAS koordiniert diese Tätigkeiten und bietet allen Prozessen über einen gemeinsam genutzten Datenspeicher eine vollständige Darstellung des Systemzustandes.

  Auch hier ist wieder eine gewisse Verwandschaft zu den Komponentenmodellen vorhanden, obwohl MANDAS wohl eher aus einer betriebssystem-orientierten Sicht heraus entstanden ist.

- **Meta**

  Meta von der Cornell University in New York versteht sich als eine Infrastruktur zur fehler-toleranten Verwaltung von verteilten Anwendungen, wobei eine verteilte Anwendung eine Menge von kooperierenden Prozessen darstellt, die untereinander Nachrichten austauschen.

  Der integrale Bestandteil von *Meta* ist die *Reactive System Architecture*, die auf von den Prozessen ausgelöste Ereignisse reagiert. Die Regeln, nach denen eine Reaktion erfolgen soll, werden über *Management Policies* vorher festgelegt, und können auf die Prozesse zurückwirken. Interaktionen zwischen den Prozessen und der Infrastruktur werden durch eine von der Architektur vorgegebene, allgemeine Schnittstelle ermöglicht.

  Meta baut auf dem Toolkit *Isis* auf, welches die Rechner eines Netzwerkes in einer Art Pool verwaltet, um sie im Bedarfsfall zur Ausführung von Prozessen zu verwenden. *Isis* sorgt in diesem Zusammenhang für die Verteilung, den Start, die Beendigung und die Überwachung der Prozesse.

  Meta und *Isis* können als Erweiterung von Betriebssystemen für verteilte Anwendungen angesehen werden.

5.4.2 Simulationsbezogen

Die Verwaltung einer Simulation ist abhängig von dem Aufbau und der Struktur einer Simulation und damit mehr oder weniger spezifisch für jeden Simulator. Die hier vorgestellte Auswahl erhebt keinen Anspruch auf Vollständigkeit, da nicht alle in Betrieb befindlichen Simulationsumgebungen aufgearbeitet werden können. Sie soll dem Leser viel mehr einen Eindruck vermitteln, welche Probleme in der Praxis von Bedeutung sind und wie Lösungen dafür gefunden werden konnten.

- **Joint Simulation System (JSIMS)**

  Das *Joint Simulation System* (JSIMS) der *Science Applications International Corporation* (USA) ([AW98], [DA99]) ist eine robuste und flexible Simulationsumgebung, die hauptsächlich zu Trainingszwecken eingesetzt werden soll. Sie verfolgt den Ansatz, die Simulation entsprechend den Übungszielen maßgeschneidert aus vorhandenen Komponenten zusammenzusetzen. Die Integration der Komponenten erfolgt über eine Infrastruktur, die allgemeine Funktionalitäten wie Nachrichtenaustausch, Zeitmanagement, etc. bereitstellt.

  Das zugrundeliegende Komponentenmodell beschränkt sich im wesentlichen auf die Beschreibung der Eigenschaften einer Komponente, die zur Auswahl derselben für ein gewünschtes Simulationsszenario benötigt werden.

- **Dynamic Simulation Environment (DSE)**

  Das *Dynamic Simulation Environment* (DSE) (*Reality by Design*, USA) bietet eine Softwarekomponenten-Architektur, die aus einer Simulationsumgebung nach DIS-Standard (*Distributed Interactive Simulation*) hervorgegangen ist ([BBD99]).

  Die Zielsetzung ist eine Optimierung der *High Level Architecture* (HLA), auf deren *Runtime Infrastructure* (RTI) sie aufsetzt. Als Motivation wird der hohe und zeitintensive Aufwand angeführt, der zur Koordination von *Federates* nötig ist, bevor diese in einer *Federation* miteinander kooperieren können.

  Die Lösung der Problematik besteht darin, daß *Federation Object Modell* (FOM) zu erweitern, so daß *Federates* in verschiedenen Federations eingesetzt werden können, ohne daß ein neues *Federation Object Model* erstellt oder die *Federates* verändert und neu kompiliert werden müssen. Außerdem kompensiert das *Dynamic Simulation Environment* einen der Mißstände der HLA Runtime Infrastructure, indem es maschinen- und compilerspezifische

Besonderheiten bei der Datenübertragung abfängt („data-marshaling“, „data-conversion“).

- **Collaborative Virtual Environment (COVEN)**

  Die skalierbare Netzwerk-Architektur COVEN ist im Rahmen des europäischen Projektes „Advanced Communications Technologies and Services“ (ACTS) in der Niederlande entwickelt worden ([BK00]). Sie unterstützt die Kooperation von Simulationspartnern über große Distanzen in einer gemeinsamen, virtuellen Umgebung.

  COVEN stellt Kommunikations-, Datenverteilungs- und andere grundlegende Dienste zur Verfügung und sorgt für ein konsistentes Bild der Simulation bei allen beteiligten Prozessen über eine gemeinsam genutzte Datenbank, die den Zustand der Simulation vollständig wiedergibt.

  Die Modellierung der Simulationsprozesse und die Funktionalität der Infrastruktur ist deutlich auf die Anwendung (Simulation virtueller Umgebungen) abgestimmt und enthält Aspekte, die von den allgemeinen Komponentenmodellen und auch von der *High Level Architecture* als domänen- oder anwendungsspezifisch angesehen werden.

- **HLA Foundation Classes (HFC)**

  Die *HLA Foundation Classes* (HFC) der John-Hopkins-University (USA) sollen dem Entwickler von *Federates* in einer HLA Simulation einen ähnlichen Rahmen bieten wie die *Microsoft Foundation Classes* (MFC) einem Programmierer von *Microsoft Windows* Applikationen ([Cox98]). Hintergrund ist die arbeitsaufwendige Überführung der Objektmodelle (*Simulation Object Model*, *Federation Object Model*) in ausführbare Programme und deren Integration in die Simulationsumgebung.

  Zum Zwecke der Automatisierung des Entwicklungsprozesses, der Vermeidung von Mehrfachimplementationen und einer verbesserten Qualität bietet das Rahmenwerk der *HLA Foundation Classes* (HFC) unter anderem Code-Generatoren, vordefinierte Modell-Schablonen und eine Quellcode-Datenbank.

- **Defense Evaluation and Research Agency (DERA)**

  Die *Defense Evaluation and Research Agency* (DERA) in Großbritannien beschreibt einen Ansatz, die Vorteile der Komponentenmodelle (Integrierte Entwicklungsumgebung, Berücksichtigung von Laufzeitaspekten, Flexibi-

lität, Wiederverwendung, etc.) in die *High Level Architecture* zu integrieren ([CH99]).

Ziel ist die Abkehr von monolithischen hin zu komponenten-basierten Simulationen, in denen *Federates* mit Hilfe von kommerziellen Komponententechnologien (in diesem Falle *Java Beans* von *Sun Microsystems*) entwickelt werden.

- **Computer Aided Federation Development Environment (CAFDE)**

  CAFDE ist ein *Integrated Development Environment* (IDE) der Firma *Synetics* (USA) ([GGBR98]) für den im Zusammenhang mit HLA definierten *Federation Development and Execution Process* (FEDEP). Auch hier gilt wieder die Argumentation, daß die „manuelle“ Entwicklung von HLA Objektmodellen zu kompliziert und langwierig ist und damit eine Automatisierung des Vorgangs, beispielsweise durch unterstützende Tools, unvermeidbar wird.

  Das *Computer Aided Federation Development Environment* stellt keine Einzellösung dar, sondern integriert die leistungsfähigsten Hilfsmittel in ein Rahmenwerk, das dem Entwickler durch den kompletten Lebenszyklus der Entwicklung einer Federation (Anforderungsdefinition, softwaretechnische Umsetzung, Verifikation und Validation, Ausführung) assistieren soll.

- **Distributed Simulation Exercise Construction Toolset (DiSECT)**

  DiSECT ist ein Produkt der Firma *TASC Incorporated* (USA) und verfolgt die gleichen Zielsetzungen (Automatisierung, Rationalisierung, Qualitätsverbesserung) wie das *Computer Aided Federation Development Environment* (CAFDE) (s.o). Das *Distributed Simulation Exercise Construction Toolset* ist jedoch auch für die Verwaltung von Simulationen unter Einsatz des DIS-Protokolls (*Distributed Interactive Simulation*) geeignet und bietet daher Potential, die Transition „alter“ Simulationen auf die *High Level Architecture* begleitend zu unterstützen (vergleiche [GGGH98]).

- **Multiple User Distributed Simulation (MUDS)**

  MUDS ist eine von der Tamkang University in Taiwan entwickelte, sogenannte „plug-and-play“ Umgebung für verteilte Simulationen ([Hua97]). Der Schwerpunkt liegt hier jedoch in der Analyse von Datenflüssen (Optimierung, Synchronisation, etc.) zwischen Prozessen in einer verteilten Simulation. Demzufolge werden die Prozesse als Bausteine der Simulation nur bezüglich ihres

Kommunikationsverhaltens modelliert und alle anderen Aspekte bleiben unberücksichtigt. Die *Multiple User Distributed Simulation* verwendet die HLA *Runtime Infrastructure* zur Durchführung des eigentlichen Datenaustauschs.

- **HLA Warrior**

  HLA Warrior vom *U.S. Army TRADOC Analysis Center* ([JP99]) ist ein Beispiel für das „Re-Engineering“ einer bestehenden monolithischen Simulation (*Janus*) zum Zwecke der Kooperation mit anderen Simulationen (DIS, HLA). Dazu wurde ein eigene, betriebssystemunabhängige „low-level“ Architektur entwickelt, die den Zustand der Simulation in Form einer von allen Komponenten genutzten Objekt-Datenbank verwaltet, einen Ereignisdienst zur Kommunikation anbietet und den Zeitfortschritt in der Simulation koordiniert.

  Die Simulationsaufgabe wird in Teilaufgaben zerlegt, die durch eigenständige Module umgesetzt werden. Dabei kann auf vorimplementierte Modelle und Algorithmen zurückgegriffen werden. *HLA Warrior* ist sehr stark auf die Simulation des ursprünglichen Anwendungsgebiets fokussiert (Waffensimulation) und bietet demzufolge nur eine sehr eingeschränkte Komponentensicht.

- **TIGeR**

  TIGeR ist ein Boden- und Luftfahrzeug-Simulator der Firma *Raytheon* (USA), welcher in den Bereichen Training sowie Forschung und Entwicklung zum Einsatz kommt ([Kin98]). Eine verteilte Softwarearchitektur stellt immer wieder benötigte Funktionalitäten wie Ausführungskontrolle, Interprozeß-Kommunikation und Datenbankanbindung zur Verfügung, während spezifische Komponenten anhand vordefinierter Schablonen das zu simulierende Verhalten implementieren.

  Die Modellierung der Komponenten bezieht sich rein auf die Kommunikationsschnittstelle und eine rudimentäre Ablaufsteuerung (Start, Stop, Reset, ...). Die Integration der Komponenten in die Architektur, die Auswahl der Komponenten für ein Experiment und die Gestaltung der Laufzeitumgebung muß durch den Menschen erfolgen.

- **SIMULTAAN**

  SIMULTAAN ist ein holländisches Gemeinschaftsprojekt unter Beteiligung von *TNO Physics and Electronics Laboratory*, *National Aerospace Laboratory* (NLR), *Delft University of Technology*, *Siemens Netherlands*, *Fokker Space*

BV and *Hydraudyne Systems and Engineering BV*. Die Zielsetzung ist die Bereitstellung einer Forschungs- und Entwicklungsplattform zur Durchführung von Simulationen nach dem DIS- oder HLA-Standard (vergleiche [KvGJ98]).

Die Konzeption von *SIMULTAAN* ist stark an der *High Level Architecture* orientiert, bietet aber eine Erweiterung der *Runtime Infrastructure* und der HLA Objektmodelle. Beispielsweise wird der Entwickler von der „low-level“ Programmierung in der HLA bewahrt und *SIMULTAAN* verspricht eine „plug-and-play“ Fähigkeit, die die rasche Rekonfiguration des aus Komponenten aufgebauten Simulators ermöglichen soll.

- **Joint Modeling And Simulation System (JMASS98)**

  Das *Illinois Institute of Technology Research Institute* (USA) bietet mit dem *Joint Modeling And Simulation System* ein Rahmenwerk zur Unterstützung des gesamten Lebenszyklus von Simulationsmodellen für nicht-interaktive Simulationen ([MHGM00]).

  Der Rahmen umfaßt die übliche Bereitstellung von allgemeinen Diensten wie Zeitmanagement, Ereignisdienst, Datenaustausch, etc. sowie unterschiedlichen Tools zur Unterstützung der Phasen des Lebenszyklus (Konzeption, Konfiguration, Ausführung und Ergebnisanalyse).

  Die Entwicklung von Simulationsmodulen wird durch Richtlinien und Standards vorgegeben, die die allgemeine Struktur eines Moduls und die Art und Weise, nach der der Datenaustausch zur Laufzeit erfolgen muß, definieren.

- **Simulation Execution Environment**

  Mit dem *Simulation Execution Environment* der Firma *BMH Associates* (USA) wird der Mensch in die Lage versetzt, alle Vorgänge rund um den Betrieb einer verteilten Simulation zu kontrollieren: Konfiguration und Initialisierung der Simulation auf den beteiligten Rechnern, Überwachung der Ausführung, Fehlerdiagnose und Ergebnisanalyse ([MB98]).

  Das *Simulation Execution Environment* dient der Automation und der Unterstützung des Nutzers, ist aber eher einem verteilten Betriebssystem als einer Simulationsarchitektur gleichzusetzen, da die Entwicklung der Simulationsprozesse und die von ihnen gemeinsam zu erfüllende Aufgabe völlig außer Acht gelassen wird.

- **Distributed Simulation System - IDE (DSS-IDE)**

  Das letzte Beispiel der Firma *Virtual Prototypes* (Kanada) demonstriert erneut die Verschmelzung der komponenten-orientierten Softwaresicht mit der *High Level Architecture* ([Nau99]).

  Aufbauend auf der *Runtime Infrastructure* von HLA als Kommunikationsstruktur bietet die DDS-IDE einen kompletten Rahmen zur Unterstützung von Entwicklern einer verteilten Simulation bestehend aus eigenständigen Komponenten. Grundlage ist ein Komponentenmodell, welches die rein inhaltliche Modellierung eines realen Systems um sogenannte *Metainformationen* (vergleiche Abschnitt 5.2.2) erweitert. *Metainformationen* definieren beispielsweise, wie eine Simulationskomponente in die Architektur integriert werden muß, oder welche Laufzeitumgebung die Komponente voraussetzt.

5.5 Zusammenfassung

Grundlage für die abschließende Bewertung der vorgestellten Konzepte sind die Randbedingungen, welche in Kapitel 2.2.2 und 3.3 erläutert worden sind. Diese lassen sich im wesentlichen auf die folgenden vier Kernaussagen komprimieren:

1. Verteilte Simulation bedeutet die Kooperation von Prozessen, die mit Hilfe des Datenmanagements ([Eng01]) kommunizieren. Ein Simulationsprozeß bzw. das zugehörige Programm darf nicht angetastet werden (Black-Box-Prinzip).

2. Die Simulationsprozesse erfüllen eine gemeinsame Aufgabe, die vom Modellmanagement berücksichtigt werden muß.

3. Das Modellmanagement darf nur ein „Aufsatz“, ein Beobachter sein. Die Simulationsprogramme und auch das Datenmanagement dürfen nicht vom Modellmanagement abhängen. Ein Re-Engineering der Simulationsprogramme und des Datenmanagements im Zusammenhang mit der Entwicklung des Modellmanagements ist nicht akzeptabel.

4. Den Grundprinzipien der *Distributed Simulation Programming Architecture* (DSPA) (modular, verteilt, flexibel, skalierbar) muß Rechnung getragen werden.

Die Prozeßverwaltung auf Betriebssystemebene ist für eine Verwendung im Modellmanagement nicht geeignet, da das Betriebssystem jeden Prozeß nur für sich selbst

betrachtet. Eine Berücksichtigung der gemeinsamen Aufgabe aller Simulationsprozesse wird nicht in Betracht gezogen.

Außerdem wären die Anforderungen hinsichtlich Verteilung (das Betriebssystem ist auf die Grenzen des lokalen Rechners beschränkt) und Flexibilität (das Modellmanagement darf nicht auf das Betriebssystem zurückgreifen, da es sonst auf eine Plattform festgelegt wäre) nicht erfüllt. Bei der Betrachtung der Ausführungsanforderungen der Prozesse kann jedoch eine inhaltliche Orientierung am Betriebssystem erfolgen, da die Verwaltung und die Bereitstellung der Ressourcen eines Rechners zu den ureigenen Aufgaben eines Betriebssystems zählt.

Komponentenmodelle bieten von der Theorie her genau den richtigen Ansatz für die Aufgaben des Modellmanagements. Sie sind allgemeingültig konzipiert, können aber auf die spezielle Anwendung hin adaptiert werden. Sie berücksichtigen Aspekte wie Integration und Kombination der Einzelbausteine zu einem Ganzen als auch Installation, Konfiguration und Beobachtung der gesamten Anwendung. Zusätzlich bieten die Komponententechnologien eine integrierte Entwicklungsumgebung zur Entlastung der Entwickler und entsprechen auch den Anforderungen nach Verteilung, Flexibilität und Skalierbarkeit.

Hindernis für einen möglichen Einsatz verfügbarer Komponententechnologien wie *COM+*, *EJB* oder *CCM* ist ihr eigentlicher Vorteil, der integrale Ansatz. Die komponenten-basierte Softwareentwicklung spannt einen Bogen, der eine Komponente von der anfänglichen Idee und Konzeption bis hin zum Betrieb rundherum einhüllt. Demzufolge müßten alle Simulationsprogramme für eine Verwendung von Komponentenmodellen als Basis des Modellmanagements überarbeitet werden, was momentan einen nicht akzeptablen Arbeitsaufwand darstellen und auch die Simulationsprogramme in ein Abhängigkeitsverhältnis zum Modellmanagement bringen würde. Die Komponententechnologien besitzen zudem eine eigene Struktur zur Kommunikation in verteilten Systemen, so daß auch das Datenmanagement überarbeitet werden müßte oder zwei Kommunikationsinfrastrukturen redundant betrieben würden.

Sekundäre Gründe, die gegen eine Verwendung von *COM+*, *EJB* oder *CCM* angeführt werden können, sind der noch immer vorhandene „Overhead“ hinsichtlich Zeit und Ressourcenbeanspruchung sowie programmiertechnische Fragen. Ersteres ist für die Leistungsfähigkeit und damit der Abbildungsgüte einer interaktiven Simulation von nicht zu unterschätzender Bedeutung, da viele Simulationsprozesse (gerade bei graphischen Darstellungen) hohe Ansprüche an die Leistungsfähigkeit

von Rechnern stellen. Die Erkenntnis dieser Problematik hat dazu geführt, daß die Anbieter von Komponententechnologien sich verstärkt mit „high-performance“ und „quality-of-service“ Anforderungen auseinandersetzen ([WLS00]).

Die programmiertechnischen Probleme ergeben sich daraus, daß das *Component Object Model* (COM+) nur für die *Microsoft Windows* Plattform zur Verfügung steht und *Enterprise Java Beans* (EJB) nur die Programmiersprache *Java* unterstützt, wohingegen das Datenmanagement Schnittstellen für *C/C++* voraussetzt. Das CORBA Component Model (CCM) als die einzige Technologie, die für verschiedene Plattformen und Programmiersprachen geeignet ist, befindet sich derzeit erst in der Umsetzung und wird auch nach Fertigstellung noch einen Prozeß der Reife benötigen.

Die *High Level Architecture* formuliert eine enger gefaßte Komponentensicht für verteilte Simulationen. Die Vorteile ergeben sich aus der Betonung simulationsspezifischer Aspekte wie Zeitmanagement und optimierter Datenaustausch. Diese sind aber eher für die Performance der Simulationsprozesse und des Datenmanagements als für die Aufgaben des Modellmanagements von Bedeutung. Die Nachteile bestehen unter anderem darin, daß Fragen bezüglich der Laufzeitumgebung (Installation und Konfiguration der Komponenten, Ausführungsanforderungen, etc.) außer Acht gelassen werden.

Außerdem fehlt der *High Level Architecture* die Dynamik, die eine flexible Forschungs- und Entwicklungsplattform hinsichtlich Integration und Rekonfigurierbarkeit von Komponenten einer verteilten Simulation erwartet. Die vermehrte Entwicklung von Tools durch Benutzer der HLA schafft zwar teilweise Abhilfe, jedoch sind die Hilfsmittel nicht ohne weiteres in der verteilten Simulationsarchitektur *DSPA* einsetzbar, da sie unterschiedliche Zielsetzungen verfolgen und anderen Randbedingungen unterworfen sind.

Eine Verwendung der *High Level Architecture* würde gleichfalls ein Re-Engineering des Datenmanagements erforderlich machen, da die HLA nicht ohne die zugehörige *Runtime Infrastructure* (RTI) als Kommunikationsstruktur benutzt werden kann. Aufgrund des zu erwartenden Aufwands und der Tatsache, daß das Datenmanagement beispielsweise hinsichtlich Beobachtbarkeit der Simulation und Überwachung des Datenaustauschs Vorteile gegenüber der RTI bietet (vergleiche [Eng01]), muß diese Alternative jedoch verworfen werden. Die umgekehrte Möglichkeit, die RTI so zu erweitern, daß sie die gleiche Funktionalität wie das Datenmanagement bietet, scheitert daran, daß die *Runtime Infrastructure* nur in Binärform verfügbar ist.

Tabelle 5.1 faßt die Vor- und Nachteile kurz zusammen.

Als Konsequenz aus obiger Diskussion folgt, daß der Ansatz der Komponentenmodelle übernommen und für eine Entwicklung des Modellmanagements in der verteilten Simulationsarchitektur *DSPA* adaptiert werden muß.

Auf der Grundlage eines selbstentwickelten Komponentenmodells muß das Modellmanagement einen infrastrukturellen Rahmen aufbauen, der eine Verwaltung der Simulationsmodelle am Fachgebiet Flugmechanik und Regelungstechnik ermöglicht, dabei aber das existierende Datenmanagement und die Simulationsmodule so beläßt, wie sie sind.

Dies soll dadurch erreicht werden, daß im Gegensatz zu den existierenden Komponententechnologien inhaltliche und nicht-inhaltliche Gesichtspunkte bei der Entwicklung eines Simulationsmoduls getrennt werden. Inhaltlich meint hierbei die Umsetzung der Simulationsaufgabe, wohingegen nicht-inhaltliche Aspekte die Integration eines Simulationsmoduls in die *DSPA*, die Kombinierbarkeit mit anderen Komponenten, die Bedienung eines Simulationsprogramms sowie die Laufzeitumgebung eines Prozesses betreffen. Letzteres wird in das Modellmanagement und nicht in das Simulationsprogramm integriert werden. Die Basis für die Wahrnehmung dieser Aufgaben sind Informationen über das Simulationsmodul, welche derart gewonnen werden müssen, daß die Verwendung der Simulationsprogramme frei von Seiteneffekten bleibt. Es darf keinen Unterschied machen, ob ein Simulationsprozeß „von Hand“ und völlig eigenständig gestartet wird (beispielsweise vom Entwickler selbst) oder ob als Teil einer vom Modellmanagement verwalteten Simulation.

Die nach diesem Ansatz erfolgte Konzeption und Umsetzung eines Modellmanagements für die *Distributed Simulation Programming Architecture*, im weiteren als *Nemo's Model Organizer* bezeichnet, beschreibt das folgende Kapitel.

	Positiv	Negativ
Betriebssysteme	• Verwaltung von Ressourcen des Rechners	• Jeder Prozeß wird einzeln betrachtet • Auf lokalen Rechner beschränkt • Festlegung auf eine Plattform
Komponentenmodelle	• Unterstützung für komponenten-basierte, verteilte Anwendungen allgemein • Integraler Ansatz • Berücksichtigung aller Aspekte des Modellmanagements	• Abhängigkeit der Komponenten vom Modellmanagement • Re-Engineering der Komponenten erforderlich • „Performance Overhead“ • Programmiertechnische Probleme
High Level Architecture	• Unterstützung für komponenten-basierte, verteilte Simulation • Betonung simulationsspezifischer Aspekte (Zeitmanagement, Performance Anforderungen)	• Eingeschränkte Komponentensicht (Stichwort Laufzeitumgebung) • Fehlende Dynamik bezüglich Integration und Konfigurierbarkeit • Re-Engineering des Datenmanagements erforderlich

Tab. 5.1: Zusammenfassende Beurteilung

6 NeMO's Model Organizer

Nemo's Model Organizer (*NeMO*) ist die Bezeichnung für das Modellmanagement in der verteilten Simulationsarchitektur *DSPA*. Die Konzeption und Umsetzung von *NeMO* stellt den Kern der vorliegenden Arbeit dar.

Als Ergebnis der Recherche im vorigen Abschnitt ist festgehalten worden, daß das Modellmanagement auf der Basis der allgemeinenen Komponentenmodelle eine individuelle Modellierung der Simulationsprogramme und -prozesse in der *DSPA* (*NeMO*'s Komponentenmodell) erarbeiten muß.

Die Modellierung gibt wieder, wodurch eine einzelne Komponente identifiziert wird und welche Interaktionsmöglichkeiten zwischen den Komponenten bestehen. Desweiteren werden durch das Modell Anforderungen an die Laufzeitumgebung einer Komponente erfaßt und eine Anleitung zur Benutzung der Komponente in der Simulation bereitgestellt. Auf der Basis dieses Komponentenmodells läßt sich dann auch der Vorgang der Integration („Eincheckvorgang") dadurch charakterisieren, daß während dessen alle Informationen zur Beschreibung von Simulationsprogrammen gewonnen werden müssen.

Zentrales Merkmal von *NeMO*'s Komponentenmodell ist das zugrundeliegende Beobachter-Prinzip. Dadurch ist die Integration einer Komponente in die *DSPA* und die Verwaltung durch das Modellmanagement im Gegensatz zu gängigen Komponentenmodellen völlig frei von Seiteneffekten für die Komponente.

Als Grundlage für das Komponentenmodell wird zu Beginn dieses Kapitels ein Rechnermodell formuliert und ein Signaturbegriff für Simulationsprogramme und -prozesse in der *DSPA* eingeführt. Mit ersterem werden die Charakteristika der in der Simulation eingesetzten Rechner abgebildet sowie deren Leistungsfähigkeit und Belastung ähnlich dem Prinzip des *„Benchmarking"* erfaßt. Letzteres ermöglicht die Identifikation und Unterscheidung von Programmen und Prozessen anhand ihres sogenannten Kommunikationsverhaltens, was eine notwendige Voraussetzung für einen Vergleich und eine Bewertung ihrer Aufgabe in der Simulation darstellt.

Nemo's Model Organizer als Bestandteil des Rahmenwerks der verteilten Simulationsarchitektur *DSPA* ist selbst eine verteilte Anwendung. Um zu verstehen, wie es seine Aufgaben wahrnehmen kann, wird dessen interne Baum-Struktur und die Aufteilung in die drei eigenständigen Bestandteile (*Local*, *Shared* und *Central Nemo*) vorgestellt werden.

Am Ende des Abschnitts wird auf Details der Implementation eingegangen werden.

Oberstes Ziel war und ist die Plattformunabhängigkeit des Modellmanagements. Welche Maßnahmen ergriffen worden sind, um das für das Ergebnis dieser Arbeit als auch für die Zukunft zu gewährleisten, wird dort beschrieben.

6.1 Rechnermodell

Im Zuge der Verwaltung der Simulationsprogramme durch das Modellmanagement können die Rechner, auf denen die Programme ausgeführt werden, nicht außer Acht gelassen werden, da zwischen den Prozessen und den Computern eine direkte Abhängigkeit besteht (vergleiche Kapitel 4.1).

Nemo's Model Organizer formuliert aus diesem Grund ein vereinfachtes Rechnermodell, welches prinzipiell

1. die Anforderungen von Simulationsprozessen berücksichtigt,
2. die Verwaltung der zur Verfügung stehenden Ressourcen ermöglicht, sowie
3. Aussagen über Leistungsfähigkeit bzw. Belastung der Computer zuläßt.

Der Ansatz besteht darin, eine abstrakte, idealisierte und betriebssystemunabhängige Beschreibung von Rechnern zu finden, mit deren Hilfe eine Art „Vorverwaltung" („Pre-Processing") der Prozeßanforderungen durch das Modellmanagement erfolgen kann.

„Vorverwaltung" bezieht sich darauf, daß die eigentliche Verwaltung der Ressourcen eines Rechners immer noch durch das zum Rechner gehörige Betriebssystem wahrgenommen wird. Alles andere wäre kontraproduktiv hinsichtlich *NeMO*'s Zielsetzung nach Plattformunabhängigkeit, da das Modellmanagement dann geräte- und betriebssystemspezifisches Wissen zur Erfüllung seiner Aufgaben benötigen würde. Außerdem ist die in der Simulation eingesetzte Hardware einem ständigen Wandel unterworfen, so daß es allein schon deswegen ein naives Ansinnen wäre, den Einsatz aller möglichen technischen Geräte vorherzusehen und deren Charakteristika in das Modellmanagement mit aufzunehmen.

In diesem Sinne kann man *NeMO* als eine Koordinationsstelle ansehen, wo alle Anforderungen der Prozesse gesammelt und einer Eingangsprüfung unterzogen werden. Nur wenn ein Simulationsprogramm die Berechtigung besitzt, auf einem bestimmten Rechner gestartet zu werden, und dieser noch genügend freie Ressourcen zur

Ausführung desselben bereitstellen kann, wird das Programm an das Betriebssystem übergeben. Von da an ist die Benutzung des Rechners wie üblich eine reine Angelegenheit zwischen dem Simulationsprogramm und dem Betriebssystem.

Die Beurteilung der Leistungsfähigkeit bzw. der Belastung von Computern wird auf einen Zahlenvergleich zurückgeführt, ähnlich dem Prinzip des „Benchmarking“. Die Begründung hierfür ist wiederum die Unabhängigkeit des Modellmanagements von den spezifischen Eigenheiten der Hardware und des Betriebssystems. Da eine Charakterisierung von Computer-Bausteinen allgemein mit Hilfe absoluter Zahlenwerte keinen Sinn ergibt, werden Referenzbausteine definiert. Durch einen Faktor größer oder kleiner eins kann dann für jeden anderen Baustein angegeben werden, wie gut oder schlecht er im Vergleich zum Referenzobjekt ist (relative Bewertung). Die Festlegung der Referenz kann bei Veränderungen des Rechnernetzwerks angepaßt werden.

6.1.1 Repräsentation

Abbildung 6.1 zeigt den vereinfachten, schematischen Aufbau eines Rechners aus der Sicht von *Nemo's Model Organizer* (vergleiche auch [Kub96]). Es lassen sich zwei eigenständige Einheiten erkennen:

- Der Kern des Rechners (Prozessor/en, Speicher).
- Die Ein- und Ausgabegeräte.

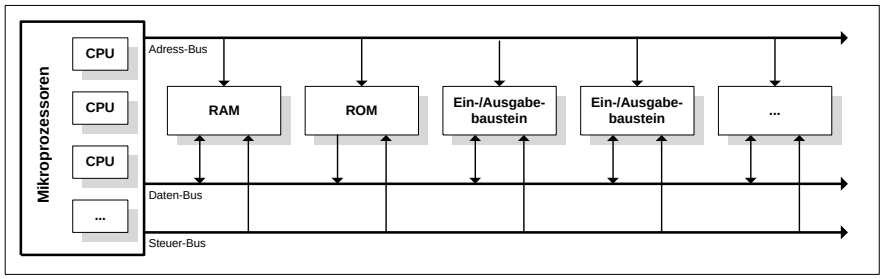


Abb. 6.1: Idealisierter Rechneraufbau

Das Betriebssystem als Hülle um den Rechnerkern und die I/O-Bausteine kann über seinen *Namen* und seine *Versionsnummer* eindeutig unterschieden werden. Weitergehende Informationen als die zur Unterscheidung werden von *Nemo's Model Organizer* nicht benötigt.

Der Kern des Rechners umfaßt den Teil der Rechnerarchitektur der als minimal für die Ausführung eines Prozesses angesehen werden kann. Dazu gehören ein oder mehrere Steuerungseinheiten (Prozessoren), der Hauptspeicher (*Read Only Memory* und *Random Access Memory*) sowie die verschiedenen Bussysteme (Adress-Bus, Daten-Bus, Steuer-Bus). Die wesentlichen Eigenschaften eines Rechnerkerns, vom Standpunkt des Modellmanagements aus betrachtet, faßt Tabelle 6.1 zusammen.

Attribut	Art	Beschreibung
Identifier	Zeichenkette	Name des Rechnerkerns
Processor Number	Ganzzahl	Anzahl der vorhandenen Prozessoren
Processor Type	Zeichenkette	Bezeichnung für den Prozessorentyp
Processor Rating	Fließkommazahl	Leistungsfähigkeit des Prozessorentyps im Vergleich zu einem Referenztyp
Memory Amount	Ganzzahl	Menge des zur Verfügung stehenden Hauptspeichers
Load	Fließkommazahl	Momentane Belastung des Rechnerkerns (Zahl zwischen Null und Eins)
Connected Requests	Liste	Liste mit Anforderungen der momentan von diesem Rechnerkern ausgeführten Prozesse

Tab. 6.1: Repräsentation des Rechnerkerns

Das Betriebssystem und der Kern des Rechners müssen notwendigerweise für jeden Computer, der vom Modellmanagement verwaltet werden soll, beschrieben werden.

Der Begriff Ein- und Ausgabegeräte summiert alle Bausteine, die der Kommunikation mit dem Rechner dienen. Beispiele für die momentan am Fachgebiet Flugmechanik und Regelungstechnik in der Simulation eingesetzten Geräte sind wie folgt:

- Mouse
- Tastatur

- LCD/Monitor
- Festplatten
- Projektoren für die Außensicht
- Lautsprecher
- Flight Control Unit (FCU) als Hardware-In-The-Loop
- Analoge Eingabegeräte über Wandlerkarten
- Head Mounted Display
- Helmet Mounted Display
- Head-Up-Display
- Positions- und Lage-Tracker

Die vermeintlich große Vielfalt läßt sich bei einer Betrachtung der Art der Signalübertragung auf eine übersichtlichere Zahl reduzieren. Da es für einen Rechner keinen Unterschied macht, ob an seinem Video-Ausgang ein LCD oder ein Monitor (CRT) angeschlossen ist, soll der Modellierung der Ein- und Ausgabemöglichkeiten eher eine rechnerinterne Sicht zugrundeliegen, die nach den verwendeten I/O-Controllern differenziert:

- Serielle Schnittstellen (PS/2, RS-232, ...):

  Maus, Tastatur, FCU, Tracker

- Parallele Schnittstellen (VME-Bus, SCSI, IDE, ...):

  Wandlerkarten, Festplatten

- Bildsignale (RGB, VGA, ...):

  LCD, Monitor, Projektor, Head Mounted Display, Helmet Mounted Display, Head-Up-Display

- Tonsignale (WAV, AVI, ...):

  Lautsprecher

Als Ergebnis der Generalisierung definiert *Nemo's Model Organizer* eine einheitliche Repräsentation aller Ein- und Ausgabegeräte, welche in Tabelle 6.2 vorgestellt wird. Die Beschreibung der I/O-Bausteine eines Rechners ist optional. Es müssen nur diejenigen dem Modellmanagement bekannt gemacht werden, die in der Simulation benötigt und auch verwendet werden.

Attribut	Art	Beschreibung
Identifier	Zeichenkette	Name des I/O-Bausteins
Unit Type	Zeichenkette	Bezeichnung für den Typ des I/O-Bausteins (Video, Audio, Serial, Parallel)
Remote I/O	Wahrheitswert	Schalter zur Festlegung, ob dieser Baustein auch von einem anderen Rechner aus erreichbar ist
Number	Ganzzahl	Anzahl der vorhandenen Instanzen des I/O-Bausteins
Name List	Liste	Liste mit den Namen der verfügbaren Instanzen
Instance Rating	Fließkommazahl	Leistungsfähigkeit einer Instanz dieses Bausteins
Load	Fließkommazahl	Momentane Belastung des Bausteins (Zahl zwischen Null und Eins)
Connected Requests	Liste	Liste mit Anforderungen der momentan ausgeführten Prozesse, die diesen Baustein benutzen

Tab. 6.2: Repräsentation der Ein- und Ausgabebausteine

6.1.2 Anforderungen

Prozesse beschreiben die Voraussetzungen für ihre Ausführung in Form von Anforderungen, die sich naturgemäß an den Repräsentationen des Rechnerkerns und der Ein- und Ausgabegeräte orientieren (siehe Tabellen 6.3 und 6.4).

Attribut	Art	Beschreibung
Identifier	Zeichenkette	Bezeichnung des angeforderten Rechnerkerns
Identical Match	Wahrheitswert	Schalter zur Festlegung, ob exakt dieser Rechnerkern benötigt wird oder ob auch einer mit äquivalenter Leistung akzeptiert wird
Processor Number	Ganzzahl	Anzahl der angeforderten Prozessoren
Memory Amount	Ganzzahl	Menge des angeforderten Hauptspeichers
Using Policy	Auswahl	Art der Nutzung der Prozessoren (gemeinsam, exklusiv)

Tab. 6.3: Definition der Anforderung an einen Rechnerkern

Attribut	Art	Beschreibung
Identifier	Zeichenkette	Bezeichnung des angeforderten Bausteins
Unit Type	Zeichenkette	Typ des angeforderten Bausteins
Identical Match	Wahrheitswert	Schalter zur Festlegung, ob exakt dieser Baustein benötigt wird oder ob auch einer mit äquivalenter Leistung akzeptiert wird
Number	Ganzzahl	Anzahl der angeforderten Instanzen dieses Bausteins
Using Policy	Auswahl	Art der Nutzung der Instanzen (gemeinsam, exklusiv)

Tab. 6.4: Definition der Anforderung an Ein- und Ausgabebausteine

In der Formulierung der Anforderungen sind Freiheitsgrade möglich, die sich hinsichtlich Kern und I/O-Bausteine nur unwesentlich unterscheiden. Ein Prozeß kann seine Anforderung an einen Rechnerkern vollständig undefiniert lassen, wohingegen er hinsichtlich Ein- und Ausgabegeräte zumindest den Typ der Bausteine in seiner Anforderung festlegen muß.

Der *Identifier* kann in beiden Fällen undefiniert gelassen werden, was gleichbedeutend damit ist, daß ein beliebiger Rechnerkern bzw. Baustein den Ansprüchen des Prozesses genügt. Wird jedoch ein Name angegeben, ist der Prozessortyp bzw. Bausteintyp und die entsprechende Leistungsfähigkeit eindeutig bestimmt. In diesem Fall kann mit Hilfe des Wahrheitswertes *Identical Match* ausgewählt werden, ob exakt der beschriebene Kern oder Baustein dem Prozeß zur Verfügung gestellt werden muß oder ob auch ein anderer mit äquivalenter Leistungsfähigkeit akzeptiert wird.

Mit *Processor Number* und *Memory Amount* äußert ein Prozeß, wieviel Prozessoren und wieviel Hauptspeicher er benutzen möchte. Bei fehlenden oder ungültigen Angaben wird immer vom Minimum Eins (ein Prozessor, eine Einheit Hauptspeicher) ausgegangen. Bezüglich Ein- und Ausgabegeräte kann die Anzahl (*Number*) der gewünschten Instanzen mitgeteilt werden.

In der Art und Weise der Nutzung (*Using Policy*) von Rechnerkernen und den Bausteinen wird unterschieden, ob ein Prozeß diese exklusiv beansprucht („exclusive“) oder ob er bereit ist, sie mit anderen zu teilen („shared“). Die Voreinstellung ist immer die gemeinsame Benutzung von Ressourcen.

6.1.3 Belastung und Erfüllungsgrad

Die Belastung eines Rechnerkerns oder eines Bausteins ergibt sich aus der Summe der Anforderungen, die von Prozessen an die Ressourcen gestellt werden. Da der Schwerpunkt dieser Arbeit nicht auf der Modellierung der Leistungsfähigkeit und Beanspruchung eines Computers liegen soll, wird die Belastung einer Einheit (Kern oder Ein-/Ausgabe) auf der Basis eines einfachen additiven Modells errechnet und durch eine Zahl größer, kleiner oder gleich Eins ausgedrückt (siehe Gleichung 6.1). Ein Wert größer Eins ist gleichbedeutend mit einer Überbelastung, ein Wert kleiner Eins besagt, daß die Einheit noch Leistungsreserven besitzt, und der Wert gleich Eins wird im Falle einer optimalen Ausnutzung der Leistungsfähigkeit der Einheit erreicht.

$$Belastung = \frac{\sum Anforderungen}{Leistungsfähigkeit} \tag{6.1}$$

Hierbei gilt:

$$\begin{array}{lcll} Belastung & < & 1.0 & : \quad Unterlast \\ Belastung & = & 1.0 & : \quad optimale \ \ Ausnutzung \\ Belastung & > & 1.0 & : \quad Überlast \end{array}$$

Aus der Sicht eines Prozesses ist jedoch nicht die Gesamtbelastung des Bausteins durch alle Prozesse von Bedeutung, sondern vielmehr eine Aussage darüber, wie gut der Rechnerkern oder das Ein-/Ausgabegerät die individuelle Anforderung erfüllen kann. Dazu wird ein sogenannter Erfüllungsgrad definiert (siehe Gleichung 6.2). Alle Anforderungen der mit einem Baustein verbundenen Prozesse werden zur Berechnung des Erfüllungsgrades gleich stark gewichtet, so daß dieser nichts anderes als der Umkehrwert der Belastung ist. Der Erfüllungsgrad kann im Gegensatz zur Belastung per Definition nie größer als Eins werden.

$$Erfüllungsgrad = \frac{Leistungsfähigkeit}{\sum Anforderungen} \leq 1.0 \tag{6.2}$$

6.2 Signaturbegriff

Ausgehend von der Analyse des Simulationsmodells wurde in Kapitel 4.2 die Notwendigkeit eines Signaturbegriffs für das Modellmanagement formuliert. Erst mit Hilfe einer „Kennzeichnung“ (wörtliche Übersetzung des lateinischen Fremdwortes Signatur) können die Simulationsprozesse während einer Simulation identifiziert und unterschieden werden. Damit ist dann eine Grundlage geschaffen, um das Verhalten der Prozesse vergleichen, bewerten oder beurteilen zu können.

Wie der allgemeine Begriff der Signatur für die Belange des Modellmanagements adaptiert und spezifiziert werden muß, soll im folgenden erläutert werden.

6.2.1 Grundlagen

Ausgangspunkt der Adaption des Signaturbegriffs ist die ursprüngliche Definition ([NIS]):

> **Signature:** „The types or domains, and order, in some representations, of inputs to and outputs from a function."

Die Definition für *domains* soll zum besseren Verständnis hier ebenfalls zitiert werden:

> **Domain:** „The input, for which a function or relation is defined, ... the possible values of a variable."

Die Betrachtung der Signatur erstreckt sich also ähnlich wie die Komponentensicht (vergleiche Abschnitt 5.2) nur auf die Schnittstellen, die von außen erkennbar sind.

Als einzige Schnittstelle der Prozesse in der Simulation, die unabhängig von der eigentlichen Prozeßaufgabe und auch im Rahmen der verteilten Simulationsarchitektur *DSPA* unveränderlich ist, bleibt nur die Verbindung der Prozesse zum Datenmanagement. Diese ist für alle Prozesse fest vorgegeben und stellt die einzige Möglichkeit dar, an einer Simulation teilzunehmen.

Aus diesem Grund kann nur die Schnittstelle zum Datenmanagement als Grundlage für *NeMO*'s Signaturbegriff herangezogen werden. Zusätzlicher Vorteil ist hierbei, daß diese Schnittstelle ebenfalls mit Hilfe des Datenmanagements beobachtet werden kann (vergleiche [Eng01]), so daß Informationen über die „Inputs" in und die „Outputs" von einem Simulationsprozeß auf automatisierte Art und Weise gewonnen werden können und nicht vom Menschen erfragt werden müssen.

Eine nähere Betrachtung der Interaktion der Simulationsprozesse mit dem Datenmanagement läßt eine Zweiteilung in der Verwendung der Schnittstelle erkennen ([Eng01]).

1. **Anmeldung**

   Nachdem ein Prozeß mit dem Datenmanagement in Kontakt getreten ist, meldet er zwei Listen mit Namen beim „Datenpool" an; die eine für Daten, die der Prozeß lesen möchte, die andere für die zu schreibenden Daten. Mit den

Namen werden einerseits Attribute wie Datentyp (*float*, *integer*, ...) und Voreinstellung (*defaults*), andererseits ein Wert verbunden. Die Summe der Eigenschaften und der zugehörige Wert beschreiben ein Objekt, welches vom Datenmanagement als *Alias* bezeichnet wird (vergleiche [Eng01]).

2. **Kommunikation**

   Nach der Initialisierungsphase beginnt die Abarbeitung der eigentlichen Aufgabe des Prozesses, während dieser er in einer Schleife („main-loop“) immer wieder drei Arbeitsschritte durchläuft: Daten lesen, Daten verarbeiten, Daten schreiben (vergleiche Abbildung 4.3).

   In dieser Phase werden zwischen dem „Datenpool“ und dem Prozeß sogenannte Name-Wert-Paare („Named-Values“) ausgetauscht. Wie die Bezeichnung schon erahnen läßt, wird ein Wert für einen *Alias* (s.o.) kommuniziert, wobei der *Alias* über den Namen eindeutig identifiziert wird. Außer der Angabe, ob der Prozeß einen Wert lesen oder schreiben möchte, werden zu diesem Zeitpunkt keine weiteren Informationen vom Datenmanagement benötigt.

Im folgenden soll der Begriff *Kommunikationsverhalten* verwendet werden, der hiermit definiert wird:

> Das **Kommunikationsverhalten** eines Prozesses umfaßt die Anzahl und die Art der ausgetauschten *Aliase* („Read-Alias“ oder „Write-Alias“).

Prozesse sind die Instanzen von Programmen, wobei alle Instanzen eines Simulationsprogramms nicht notwendigerweise das gleiche Kommunikationsverhalten aufweisen müssen. Das Gegenteil ist vielmehr der Fall; das Verhalten eines Prozesses kann meist mit Hilfe von Parametern, die dem Programm zur Startzeit übergeben werden, flexibel und dynamisch angepaßt werden.

Demzufolge kann ein Programm als Kodierung einer Menge von möglicher Kommunikationsverhaltensformen betrachtet werden, wohingegen ein Prozeß nur ein definiertes Kommunikationsverhalten aufweist (siehe Abbildung 6.2).

Die Information über das Kommunikationsverhalten der Prozesse ist wesentlich für die Aufgabenerfüllung des Modellmanagements und muß in diesem Zusammenhang auch dauerhaft gespeichert werden können (Stichwort Sollverhalten und Vergleichskriterien). Es ist jedoch nicht angebracht, alle möglichen Verhaltensformen von Prozessen (Instanzen eines Programms) explizit zu bestimmen und abzuspeichern.

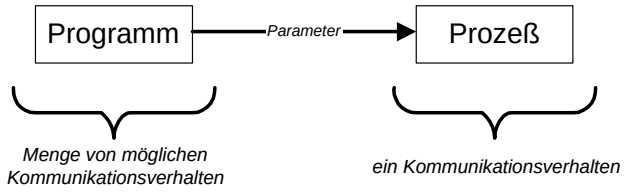


Abb. 6.2: Parametrisierung von Programmen

Deswegen wendet *NeMO* den Signaturbegriff nicht nur auf Prozesse, sondern auch auf die dazugehörigen Programme an. Hierbei stellen die Parameter die Eingänge in das Programm dar, mit deren Hilfe eine Instanz des Programms (Prozeß) erzeugt wird. Die Instanz, beschrieben durch ihr Kommunikationsverhalten, ist in der Begrifflichkeit der Signatur der Ausgang der Funktion „Programm".

Diese implizite Form der Beschreibung der möglichen Verhaltensformen eines Prozesses ist für jedes Programm konstant und deshalb dafür geeignet, dauerhaft (persistent) festgehalten zu werden. Zur Laufzeit kann *NeMO* dann abhängig von der getroffenen Parameterauswahl die Signatur eines Prozesses mit Hilfe der Signatur des dazugehörigen Programmes vorhersagen.

Nach dieser grundlegenden Betrachtung der Problematik wird in den folgenden Abschnitten detailliert erläutert werden, was ein *Alias* und eine *Signatur* aus der Sicht des Modellmanagements darstellt und welche Ausprägungsformen sie besitzen können.

6.2.2 Alias

Die hier formulierte Sicht eines Alias ist nicht zu verwechseln mit der Aliasdefinition des Datenmanagements ([Eng01]). Die Vorstellung des Modellmanagements baut zwar auf der des Datenmanagements auf, läßt aber einerseits gewisse Aspekte im Hinblick auf den Signaturbegriff außer Acht, andererseits wird die Definition des Alias derart erweitert, wie es für den Verantwortungsbereich von *Nemo's Model Organizer* notwendig ist.

Alias

Für die Belange des Modellmanagements wird ein Alias durch drei Eigenschaften definiert:

- Namen
- Typ (Zeichen, Ganzzahl, Fließkommazahl, etc.)
- Zugriffsform (Lesen, Schreiben)

Der Name identifiziert den Alias eindeutig, d.h. zwei Aliase sind genau dann gleich, wenn der Name identisch ist. In Erweiterung der Definition des Datenmanagements wird von *Nemo's Model Organizer* eine Namenskonvention festgelegt.

Mit Hilfe des Namens erfolgt eine Gliederung aller in einer Simulation ausgetauschten Daten in einem Baum (vergleiche Kapitel 4.4 und auch Abbildung 4.10), so daß inhaltlich zusammengehörige Informationen einfach erkennbar sind. Diese Festlegung soll auch dem Entwickler helfen, den Überblick über die Simulationsdaten und deren Beziehungen untereinander zu bewahren.

Der Name des Alias besteht aus einem *Präfix* und einem *Suffix*. Der Präfix kann wie eine Pfadangabe verstanden werden, an welcher Stelle im Datenbaum der entsprechende Alias zu finden sein wird. Der Suffix ist der eigentliche Name der Variable. Alle Aliase mit dem gleichen Präfix gehören inhaltlich zusammen, bilden in Hinblick auf das Simulationsmodell eine Einheit. Das Trennzeichen für die möglichen Ebenen des Präfix sowie für die Trennung des Suffix vom Präfix ist der Dezimalpunkt '.'. Dieses Zeichen darf nur zur Gliederung des Alias verwendet werden.

Als Beispiel sollen die Aliase, welche die drei Lagewinkel eines Flugzeugs (Hängewinkel bzw. „bank", Nicklagewinkel bzw. „pitch" und Gierwinkel bzw. „heading") repräsentieren, angeführt werden:

$$\begin{aligned} \text{Aliase:} \quad & \textit{ACR1.ATT.NAV.bank} \\ & \textit{ACR1.ATT.NAV.pitch} \\ & \textit{ACR1.ATT.NAV.head} \end{aligned}$$

Präfix und Suffix sind in diesem Falle:

$$\begin{aligned} \text{Präfix} &= \textit{ACR1.ATT.NAV} \\ \text{Suffix} &= \{\ \textit{bank}, \textit{pitch}, \textit{head}\ \} \end{aligned}$$

Die Pfadangabe durch den Präfix ist folgendermaßen zu interpretieren:

ACR1 : Abkürzung für „Aircraft“ plus Nummer
Das Flugzeug mit der Nummer Eins
ATT : Abkürzung für „Attitude“
Die Lage des Flugzeugs
NAV : Abkürzung für „Navigation“
Informationen der Navigations-Sensoren des Flugzeugs

Parametrisierter Alias

Das Kommunikationsverhalten eines Prozesses kann von Programmparametern abhängen (s.o.), d.h. erst durch Parameter wird festgelegt, welche Aliase ein Prozeß zum Lesen oder Schreiben beim Datenmanagement anmeldet. Auf welche Aliase und wie die Programmparameter sich auswirken können, wird mit Hilfe der Definition eines parametrisierten Aliases beschrieben, welche eine Erweiterung der obigen Definition eines „normalen“ Aliases ist.

Als Beispiel seien wiederum die drei Fluglagewinkel herangezogen, mit dem Unterschied, daß die Bezeichnung des Flugzeug, dessen Lage sie beschreiben sollen, dem Simulationsprogramm als Parameter übergeben werden muß.

Parametrisierte Aliase: `<<Aircraft>>`.*ATT.NAV.bank*
`<<Aircraft>>`.*ATT.NAV.pitch*
`<<Aircraft>>`.*ATT.NAV.head*

Für einen parametrisierten Alias gelten folgende Regeln:

1. Derjenige Teil des Alias, der mit Hilfe eines Parameters variabel gestaltet werden soll, wird durch den Parameternamen festgelegt. Wird der Parameter mit einem bestimmten Wert belegt, so ist der Name durch den zugehörigen Wert zu ersetzen (Name-Wert-Paar).

   Im obigen Beispiel würde demzufolge eine Belegung des Parameters `Aircraft` mit dem Wert *ACR2* folgende Aliase ergeben:

Aliase: *ACR2.ATT.NAV.bank*
ACR2.ATT.NAV.pitch
ACR2.ATT.NAV.head

2. Der Name des Parameters muß für den Menschen lesbar sein und seine Bedeutung transportieren. Eine Bezeichnung mit beispielsweise `Par1`, `Par2`, ... ist deswegen nicht gestattet.

3. Ein Alias ist mit Hilfe des Dezimalpunkts in inhaltliche Ebenen unterteilt (s.o.). Ein Parameter darf nicht ebenen-übergreifend in einen Alias eingehen.

   Das folgende Beispiel versucht zu demonstrieren, was unter ebenen-übergreifend zu verstehen ist. Gegeben seien mehrere Aliase, die die Beleuchtung („Befeuerung“) einer Runway wiedergeben sollen. Die Runway wird von einem Tower eines Flughafens verwaltet, der je nach Wetterbedingungen die Lichter ein- und ausschaltet. Die entsprechende Repräsentation durch Aliase könnte folgendermaßen aussehen:

   Aliase: *TOW1.RWY1.LIGHT.rabbit_flag*
   TOW1.RWY1.LIGHT.papi_flag
   TOW1.RWY1.LIGHT.tdz_flag
   ...

   Soll der Alias hinsichtlich der Bezeichnung des Towers und der Runway parametrisiert werden, wäre folgendes denkbar, aber nicht erlaubt.

   Aliase: `<<TowerAndRunway>>`.*LIGHT.rabbit_flag*
   `<<TowerAndRunway>>`.*LIGHT.papi_flag*
   `<<TowerAndRunway>>`.*LIGHT.tdz_flag*
   ...

   mit einer Wertebelegung des Parameters von

   `TowerAndRunway` = *TOW1.RWY1*
   `TowerAndRunway` = *TOW1.RWY2*
   ...

Eine korrekte Parametrisierung des Alias würde in diesem Falle zwei getrennte Parameter (`Tower` und `Runway`) verwenden.

Aliase: `<<Tower>>`.`<<Runway>>`.*LIGHT.rabbit_flag*
`<<Tower>>`.`<<Runway>>`.*LIGHT.papi_flag*
`<<Tower>>`.`<<Runway>>`.*LIGHT.tdz_flag*
...

4. Die Trennzeichen „`<<`“ (Beginn des Parameters) und „`>>`“ (Ende des Parameters) sind besondere Zeichen und dürfen in einem Alias außer zur Angabe, auf welche Bereiche eines Aliases sich ein Parameter erstreckt, nicht verwendet werden.

Ein parametrisierter Alias besitzt zwei Formen der Gleichheit:

- Identische Gleichheit

  Die identische Gleichheit berücksichtigt nur den Namen des parametrisierten Alias selbst. Ist dieser String gleich, so sind die parametrisierten Aliase identisch gleich.

- Bedingte Gleichheit

  Die bedingte Gleichheit zieht mögliche oder auch vorgegebene Wertebelegungen der Parameter mit in Betracht. Danach sind zwei parametrisierte Aliase bedingt gleich, wenn sich Werte für die Parameter finden lassen, so daß die parametrisierten Aliase in Übereinstimmung gebracht werden können.

  Aliase: `<<Aircraft>>`.*COP.PNL.MIP.*`<<Display>>`.*altitude_flag*

  `<<Aircraft>>`.*COP.PNL.MIP.PFD1.altitude_flag*

  Die beiden Aliase können zur Deckung gebracht werden, wenn für den ersten Alias der Parameter `Display` mit dem Wert *PFD1* belegt wird, und beide Aliase den gleichen Wert hinsichtlich des Parameters `Aircraft` verwenden. Daher sind die Aliase aus obigen Beispiel bedingt gleich.

Konflikte

Die Definition von Konflikten zwischen Aliasen orientiert sich an denen des Datenmanagements ([Eng01]).

1. Kollisionen

   Eine Kollision bedeutet die mehrfache Anmeldung eines Alias zum Schreiben von verschiedenen Prozessen. Dies ist ein schwerwiegender Konflikt und muß auf jeden Fall zu vermeiden versucht werden, da ansonsten die Eindeutigkeit des Ursprungs einer Information in der Simulation nicht mehr gewährleistet ist.

   Dadurch daß dann eventuell inkonsistente Daten im Datenpool enthalten sind, wird nicht nur einer der Prozesse, die den Alias schreiben möchten, beeinträchtigt, sondern auch alle Abnehmer (lesende Prozesse) des Alias und damit, infolge von Fehlerfortpflanzung, die gesamte Simulation.

2. Singles

   Ein Alias kann auch betrachtet werden als Verbindungsstück zwischen mindestens zwei Prozessen, einem Leser und einem Schreiber. Fehlt einer dieser „Anschlüsse“ des Alias, so bezeichnet das Datenmanagement diesen Alias als *Single*.

   In Erweiterung des Datenmanagements nimmt *NeMO* eine weitergehende Unterscheidung von Singles vor:

   - Read-Single
   - Write-Single

   Im ersten Fall wird ein Alias von einem Prozeß zum Lesen angemeldet, aber kein anderer Prozeß beabsichtigt diesen Alias zu schreiben. Der Write-Single beschreibt den umgekehrten Fall.

   Beides sind minderschwere Konflikte, dienen jedoch der Analyse des Simulationsmodells. Ein Read-Single ist ein Hinweis dafür, daß ein Simulationsmodell unvollständig ausgearbeitet worden ist (zu wenig Information) oder daß nicht alle Simulationsprozesse, die zu einem Simulationsmodell gehören, gestartet worden sind.

Durch ein Write-Single wird angedeutet, daß das Simulationsmodell nicht benötigte Informationen enthält (zu viel Information) oder daß nicht alle Prozesse eines Simulationsmodells zur Ausführung gebracht werden.

6.2.3 Signatur

Die Signaturdefinitionen stellen das Ziel der vorherigen Diskussion und Analyse dar. Sie beschreiben aus der abstrahierten Sicht des Modellmanagements heraus die Ein- und Ausgänge von Simulationsprozessen bzw. -programmen. Es sei darauf hingewiesen, daß die nun vorgestellten Definitionen immer im Zusammenhang mit den Aliasdefinitionen des vorangegangenen Abschnitts gesehen werden müssen.

Prozeßsignatur

Funktion : Ein Prozeß verknüpft Aliase (siehe Abbildung 6.3).

Eingang : Eine beliebig lange Liste von Namen der vom Prozeß gelesenen Aliase.

Ausgang : Eine beliebig lange Liste von Namen der vom Prozeß geschriebenen Aliase.

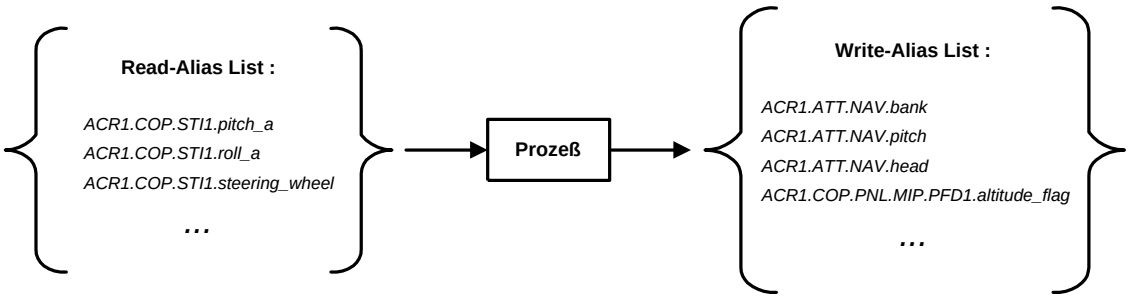


Abb. 6.3: Prozeßsignatur

Über das Innere eines Prozesses liegt kein explizites Wissen vor (Black-Box), um die Unabhängigkeit des Modellmanagements vom Simulationsmodell zu garantieren. Deswegen assoziiert *Nemo's Model Organizer* die Aufgabe eines Prozesses mit dessen Kommunikationsverhalten, d.h. zwei Prozesse, die die gleichen Aliase lesen und schreiben, erfüllen aus der Sicht von *NeMO* die gleiche Aufgabe.

Das Kommunikationsverhalten eines Prozesses kann unabhängig vom Prozeß über das Datenmanagement bestimmt werden. Aufgrund der Assoziation des Verhaltens mit der Aufgabe eines Prozesses und der Annahme, daß die Aufgabe eindeutig von den Aufgaben der anderen Prozesse getrennt ist, liegt somit ein Kriterium zur Unterscheidung bzw. Kennzeichnung der Prozesse eines Simulationsmodells vor.

Die Kenntnis über das Kommunikationsverhalten eines Prozesses ist transient, d.h. das Wissen existiert nur zur Laufzeit (siehe dazu die folgenden Ausführungen zur Programmsignatur).

Programmsignatur

Funktion	:	Ein Programm erzeugt Prozesse und spezifiziert deren Kommunikationsverhalten (siehe Abbildung 6.4).
Eingang	:	Eine beliebig lange, aber bekannte Liste von Parametern (Name-Wert-Paare).
Ausgang	:	Das Kommunikationsverhalten des Prozesses, der mit den übergebenen Parametern gestartet wird.

Das Programm stellt aus der Sicht des Modellmanagements eine sogenannte *White-Box* dar (vergleiche [Bäu98]). Eine White-Box ist dadurch gekennzeichnet, daß Wissen über das Innere vorhanden ist. In diesem Fall besteht es darin, daß *NeMO* vorhersagen kann, wie die Programmparameter das Kommunikationsverhalten des zu startenden Prozesses beeinflussen, und zwar in Form zweier parametrisierter Alias-Listen (siehe Abbildung 6.5).

Die Abbildung der Funktionalität eines Programms mit Hilfe von zwei Listen entspricht natürlich nicht dem wirklichen, vom Entwickler festgelegten Programmablauf. Es ist vielmehr eine Modellvorstellung, in der einzig das Kommunikationsverhalten des Prozesses von Bedeutung ist; wird dies der Realität entsprechend vorhergesagt, dann ist die Abbildung des Programms für die Zwecke des Modellmanagements ausreichend.

Die zwei parametrisierten Alias-Listen stellen eine vom Simulationsmodell unabhängige Beschreibung eines Programms dar, sie sind in bezug auf ein Programm unveränderlich und sie enthalten implizit die Anzahl und Namen der Parameter

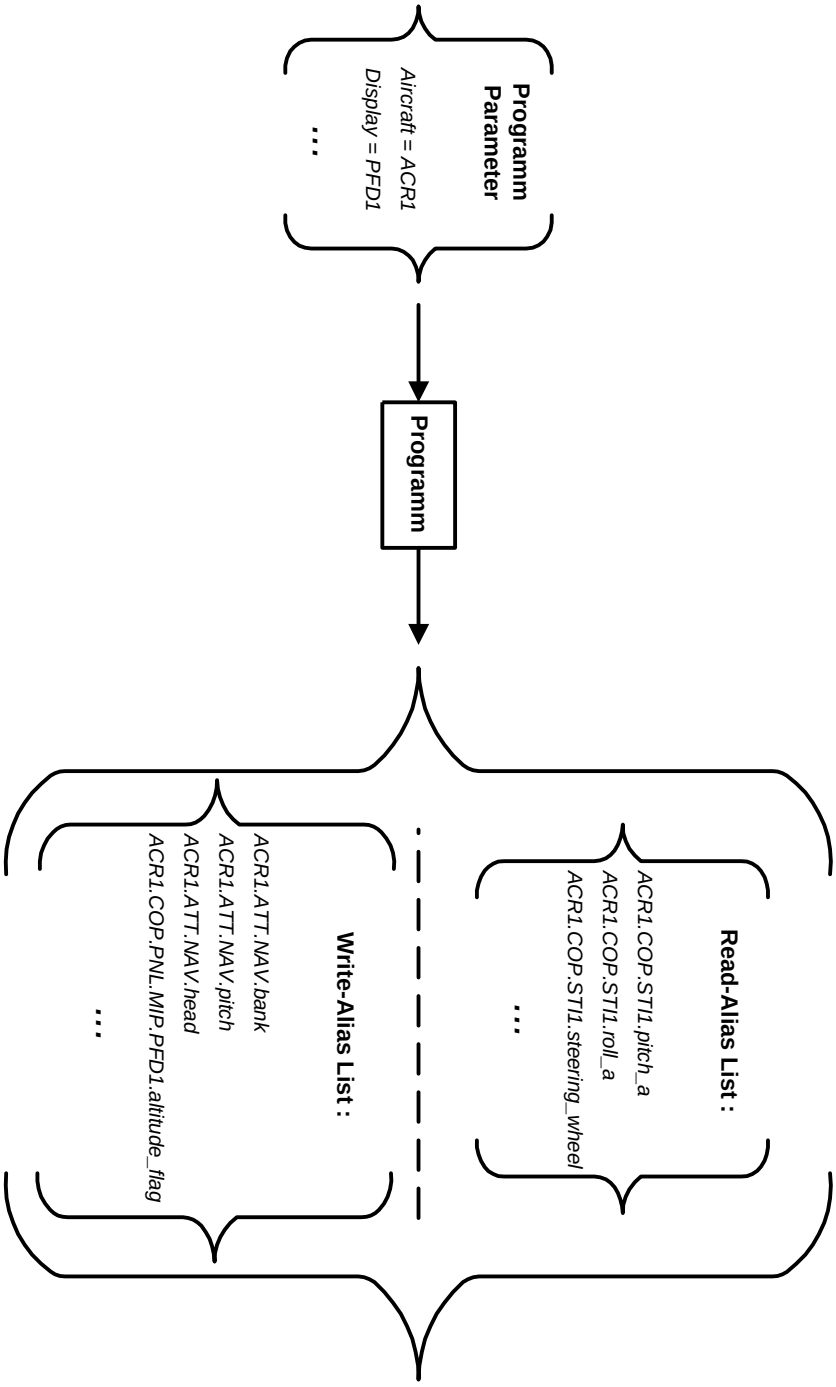


Abb. 6.4: Programmsignatur

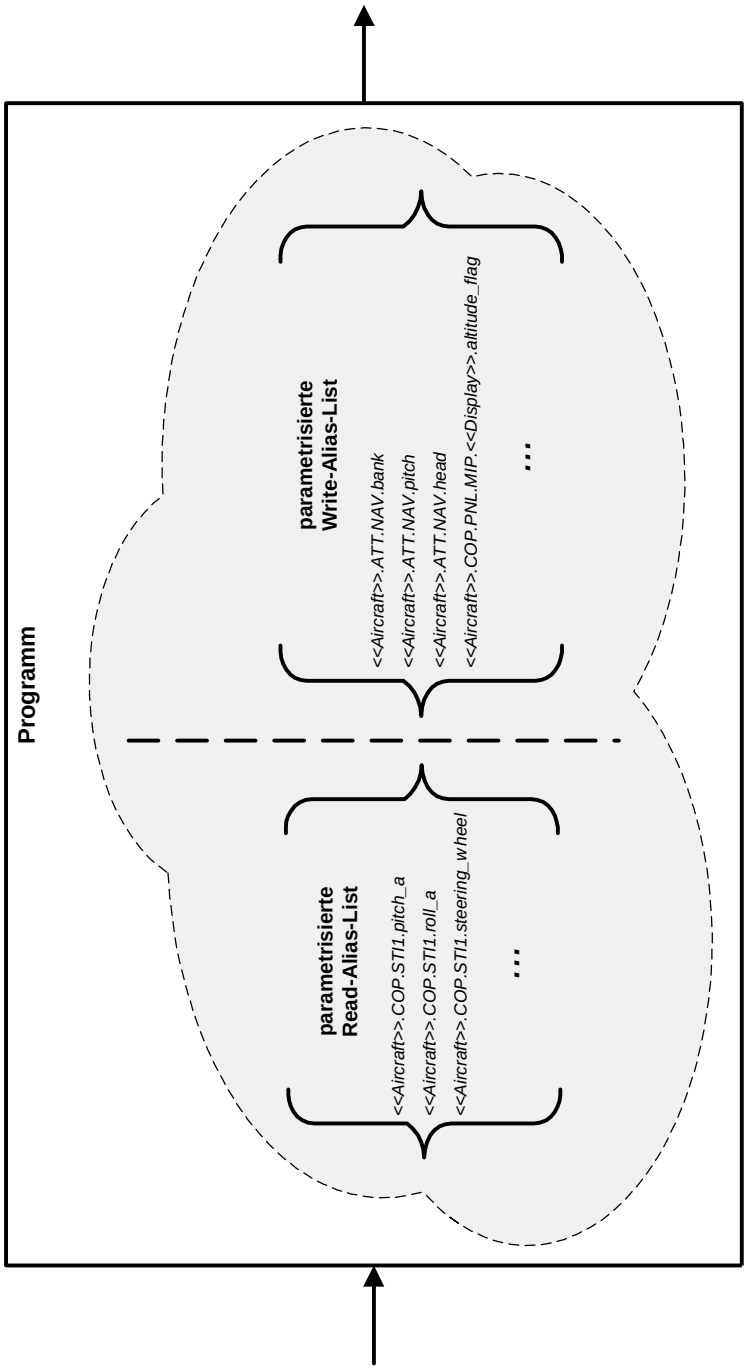


Abb. 6.5: Programmfunktion aus der Sicht des Modellmanagements

des Programms. Damit sind sie idealerweise dafür geeignet, dauerhaft (persistent) festgehalten zu werden, denn mit ihrer Hilfe kann auf elegante und effiziente Art das Kommunikationsverhalten aller Prozesse, die aus diesem Programm hervorgehen können, abgespeichert werden. Das Kommunikationsverhalten eines Prozesses wird vorhergesagt, indem die Parameternamen in den parametrisierten Aliaslisten durch die aktuellen Werte der Parameter für diesen Prozeß ersetzt werden (z.B. `<<Aircraft>>` mit *ACR2*; vergleiche auch Abschnitt 6.2.4).

NeMO ist dadurch ebenfalls in der Lage, aus dem Kommunikationsverhalten eines Prozesses auf dessen aktuelle Parameterbelegung zu schließen. In diesem Falle müssen die Stellen in den parametrisierten Aliaslisten, wo ein Parameter codiert ist, mit den Listen der aktuell gelesenen und geschriebenen Aliasen verglichen werden, um den Parameterwert zu bestimmen.

6.2.4 Operationen

Das Modellmanagement benötigt im Hinblick auf die Signaturdefinitionen verschiedene Operationen. Welche dies sind, soll hier kurz vorgestellt werden. Der Inhalt der Darstellung umfaßt die Motivation für die Operation, die Information, die der Operation zur Verfügung gestellt werden muß sowie das Ergebnis der Operation. Für eine genauere Beschreibung der Implementation und der Algorithmen sei auf [Mar01] verwiesen.

Prozeßsignatur

1. Bestimmung des Kommunikationsverhaltens

   Mit Hilfe dieser Operation wird das tatsächliche, aktuelle Verhalten eines Simulationsprozesses erfragt, d.h. welche Aliase hat er zum Lesen und Schreiben angemeldet. Dabei wird auf das *Octopus System Interface* (Osif), im speziellen auf die Schnittstelle zum *Worldmanager* des Datenmanagements zurückgegriffen ([Eng01]).

Argumente	:	keine
Ergebnisse	:	Ausführender Rechner; Prozeßidentifikation (Process Id); Liste mit gelesenen Aliasen; Liste mit geschriebenen Aliasen

2. Schnittmengen

   Die Schnittmenge zweier Aliaslisten enthält die Menge von Aliasen, die den gleichen Namen besitzen. Die Betrachtung von Schnittmengen dient der Analyse des Simulationsmodells im allgemeinen und der Erkennung von Konflikten im speziellen. Ausgehend von der obigen Definition von Konflikten werden demnach paarweise die Aliaslisten (Read-Alias und Write-Alias) zweier Simulationsprozesse verglichen.

   (a) Bestimmung von Kollisionen

Argumente	:	das Kommunikationsverhalten zweier Prozesse
Ergebnisse	:	Liste von Aliasen, die von beiden Prozessen geschrieben werden

   (b) Bestimmung von Singles

Argumente	:	das Kommunikationsverhalten zweier Prozesse
Ergebnisse	:	Liste mit Read-Singles des ersten Prozesses in Bezug auf den zweiten Prozeß; Liste mit Write-Singles des ersten Prozesses in Bezug auf den zweiten Prozeß

Programmsignatur

1. Bestimmung einer Programmabbildung

   Die Ermittlung der Abbildung eines Programms ist ein zentraler Bestandteil des Modellmanagements, da mit ihrer Hilfe die Möglichkeit besteht, das Kommunikationsverhalten eines Programms und damit aller dazugehöriger Prozesse abzuspeichern bzw. vorherzusagen. Für *Nemo's Model Organizer* ist sie deshalb ähnlich bedeutend wie beispielsweise die Eigenwerte zur Beschreibung der Schwingungseigenschaften eines dynamischen Systems.

   Im Vergleich zu den anderen Operationen ist die Bestimmung der parametrisierten Aliaslisten die weitaus komplexeste, da dafür keine eindeutige Funktion existiert, sondern auf Hilfsmittel wie Mustererkennung und Kombinatorik

zurückgegriffen werden muß. Ziel ist, aus dem aufgezeichneten Kommunikationsverhalten eines Prozesses und der Kenntnis der Parameter, mit denen der Prozeß erzeugt wurde, eine Beziehung zu formulieren, die eindeutig wiedergibt, wie die Parameter eines Prozesses sich auf dessen Verhalten auswirken.

Die Aufzeichnung des Kommunikationsverhaltens geschieht ursprünglich während des Eincheckvorgangs (siehe Kapitel 6.3.4). Sie ist jedoch aufgrund der kurzen Zeitspanne der Beobachtung nur als Momentaufnahme zu bewerten. Um die Dynamik im Kommunikationsverhalten eines Prozesses vollständig erfassen zu können, muß die Programmabbildung deswegen iterativ optimiert werden. Anstoß für eine Optimierung kann eine regelmäßige, stichprobenartige Kontrolle im Anschluß an Simulations-Versuche als auch das Auftreten von Fehlern sein.

Argumente	:	zwei Listen von Programmparametern sowie deren Werte (alle Werte müssen **eindeutig** sein); das aufgezeichnete Kommunikationsverhalten zweier Prozesse, die mit obigen Parameterbelegungen gestartet worden sind
Ergebnisse	:	parametrisierte Liste von Read-Aliasen; parametrisierte Liste von Write-Aliasen

2. Editieren einer Programmabbildung

   Das Editieren von parametrisierten Aliaslisten durch den Entwickler oder den Administrator ist als Absicherung gegenüber einer fehlerhaften automatischen Erkennung zu verstehen. Zugunsten der Prämisse, das Innere eines Simulationsprogramms bzw. eines Prozesses nicht anzutasten, wird ein möglicher Restfehler in der Abbildung eines Programms und seiner Prozesse in Kauf genommen. Durch manuelles Editieren kann dieser Restfehler beseitigt werden.

   Die Möglichkeit des Editierens beinhaltet implizit die Notwendigkeit zur Darstellung der parametrisierten Aliaslisten in einer für den Menschen lesbaren Form.

Argumente : parametrisierte Liste von Read-Aliasen;
parametrisierte Liste von Write-Aliasen

Ergebnisse : Mitteilung darüber, ob alle parametrisierten Aliase regelkonform sind (s.o.)

3. Vorhersage des Kommunikationsverhaltens

   Die Vorhersage des Kommunikationsverhaltens besteht darin, daß die parametrisierten Aliaslisten durch Ersetzen der Parameternamen mit konkreten Werten in zwei Listen von „normalen“ Aliasen überführt werden. Hintergrund hierbei ist der Wunsch mögliche Konflikte zwischen Prozessen schon zu erkennen, bevor die Prozesse überhaupt gestartet worden sind. Das vorhergesagte Verhalten kann dann in die Konfiguration der Simulation mit einfließen, und zwar in der Form, welche Prozesse dürfen nicht oder welche Prozesse müssen zur Ausführung gebracht werden.

   Mit Hilfe der Vorhersage des Kommunikationsverhaltens ist auch für jeden Prozeß die Definition eines Soll-Verhaltens möglich, welches mit dem tatsächlichen (s.o.) zum Zwecke der Beurteilung verglichen werden kann.

   Argumente : Liste von Parametern mit dazugehörigen Werten

   Ergebnisse : Liste der vorhergesagten gelesenen Aliase;
   Liste der vorhergesagten geschriebenen Aliase

4. Überprüfen einer Programmabbildung

   Eine Programmabbildung kann wie oben erläutert einen Fehler enthalten. Eine dauernde Neubestimmung zur Überprüfung ist aber aufgrund des aufwendigen Algorithmus nicht zweckmäßig. Deswegen muß auf einfache Art und Weise eine Überprüfung einer Programmabbildung vorgenommen werden können.

   Die Basis einer Verifikation ist das tatsächliche Kommunikationsverhalten, welches mit Hilfe des Datenmanagements ermittelt werden kann (s.o.). Die Kenntnis der Parameterbelegungen für den betrachteten Prozeß erlaubt es dann, das Soll-Kommunikationsverhalten des Prozesses vorherzusagen. Durch einen Vergleich der jeweiligen Aliaslisten kann dann sehr schnell eine Aussage getroffen werden, ob die Programmabbildung korrekt ist. Im Falle

eines Fehlers muß erst dann die aufwendigere Adaption der Abbildung erfolgen.

Argumente	:	Liste von Parametern mit dazugehörigen Werten; zwei Aliaslisten (Read, Write)
Ergebnisse	:	Programmabbildung korrekt oder nicht korrekt; Liste von Aliasen (Read, Write) des tatsächlichen Kommunikationsverhaltens, die nicht in der Programmabbildung enthalten sind; Liste von Aliasen (Read, Write) aus der Programmabbildung, die nicht im tatsächlichen Kommunikationsverhalten erkennbar sind

5. Schnittmengen

Eine Schnittmengenbetrachtung ist teilweise schon implizit in den obigen Operationen enthalten, wo zwei Aliaslisten miteinander verglichen werden sollen. Im Gegensatz zur einfachen Schnittmengenbildung in bezug auf Prozeßsignaturen kann es sich bei den Aliaslisten hier auch um parametrisierte Listen handeln, so daß nicht nur ein identischer Vergleich, sondern auch ein bedingter Vergleich von Aliasen (vergleiche Kapitel 6.2.2) in Frage kommen kann.

Die Motivation für den Vergleich von Aliaslisten kann vielfältig sein. So kann beispielsweise für einen selektierten Prozeß (d.h. der Prozeß ist ausgewählt, die Parameter sind festgelegt, aber er ist noch nicht gestartet) dessen Konfliktpotential im Zusammenhang mit schon ausgeführten Prozessen abgeschätzt werden. Das kann derart ausgeweitet werden, daß alle Prozesse für eine Simulation selektiert und auf Kollisionen hin untersucht werden, bevor überhaupt einer gestartet worden ist. Analog könnte man auch feststellen, ob durch die Selektion eines Prozesses ein unterbrochener Datenfluß (vergleiche 4.3) geschlossen werden kann.

Die Variationsmöglichkeiten der Schnittmengenbildung aufgrund der unterschiedlichen Argumente und erwarteten Rückgaben sind relativ groß, so daß hier nur die denkbaren Unterscheidungen für den Vergleich von zwei parame-

trisierten Aliaslisten vorgestellt werden sollen.

- Vergleich einer Liste von „normalen“ Aliasen mit einer parametrisierten Liste
  - Die Parameterbelegung für die parametrisierte Liste ist vorgegeben
  - Die Werte für die Parameter der parametrisierten Liste sind frei wählbar
- Vergleich zweier parametrisierter Aliaslisten
  - Die Parameterbelegung wird für beide Listen vorgegeben
  - Die Werte für die Parameter einer Liste sind frei wählbar, die anderen sind festgelegt
  - Alle Parameter können mit beliebigen Werten versehen werden

6.3 *NeMO*'s Komponentenmodell

NeMO's Komponentenmodell ist in Anlehnung an die in der Fachwelt aktuell diskutierten Komponentenmodelle (vergleiche Abschnitt 5.2) entstanden. Im Gegensatz dazu basiert es nicht auf einer allgemein verbreiteten Kommunikationsstruktur, sondern ist auf die Verwendung im Zusammenhang mit dem Datenmanagement *Octopus* ([Eng01]) zugeschnitten. Dadurch handelt es sich um ein sogenanntes domänenspezifisches Modell ([Bäu98]) und ist in der vorliegenden Form nur für den Einsatz in der verteilten Simulationsarchitektur *DSPA* geeignet.

Da außer der Ausrichtung auf die Kommunikationsstruktur der *DSPA* das Komponentenmodell prinzipiell auf jede verteilte Anwendung mit ähnlichem Hintergrund anwendbar ist, würde eine Adaption von *Nemo's Model Organizer* in diesem Zusammenhang auch die Benutzung in anderen Architekturen ermöglichen.

6.3.1 Prinzip

Der grundlegende Unterschied zwischen den gängigen Komponentenmodellen und *NeMO*'s Komponentenmodell besteht in der Art der Komponentenumgebung (siehe Abbildung 6.6).

Erstere erzeugen eine geschlossene Schutzhülle (den *Container*) um die Komponente herum, so daß Zugriffe nur über den Container erfolgen können (Interception-Prinzip). Außerdem müssen Komponente und Container voneinander wissen, da sie direkt miteinander interagieren.

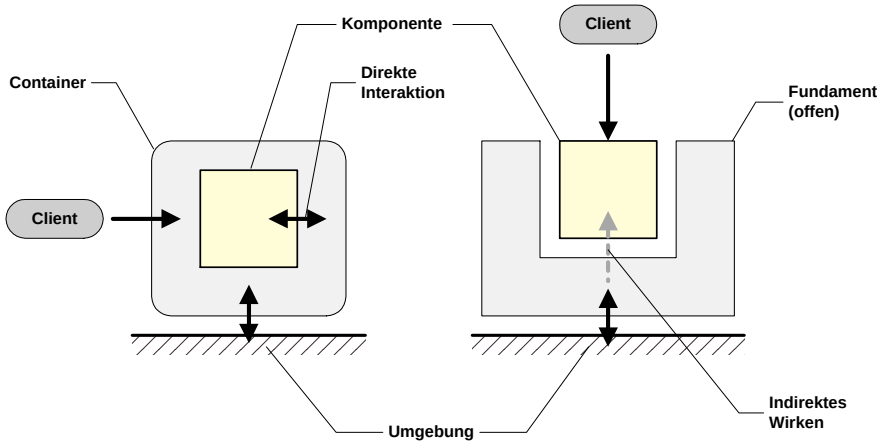


Abb. 6.6: Vergleich mit gängigen Komponentenmodellen

NeMO hingegen bietet der Komponente ein nicht geschlossenes, aber der Komponentenform angepaßtes *Fundament*, in welches die Komponente quasi eingebettet wird. Der Zugriff auf die Komponente durch Dritte kann weiterhin direkt und muß nicht über das Fundament erfolgen. Die Komponente benötigt keine Kenntnis des Fundaments, da sie keine direkten Kontakte mit dem Fundament besitzt. Das Fundament hingegen ist sich der Komponente bewußt und kann indirekt auch auf diese einwirken.

Tabelle 6.5 faßt die prinzipiellen Unterschiede zusammen.

Gängige Modelle	*NeMO*
• Geschlossene Schutzhülle	• Ein der Komponentenform angepaßtes Fundament
• Zugriff auf Komponente über Container	• Direkter Zugriff auf Komponente
• Komponente und Container interagieren miteinander	• Fundament kennt Komponente, aber nicht umgekehrt • Fundament wirkt indirekt auf Komponente

Tab. 6.5: Prinzipielle Unterschiede zu gängigen Komponentenmodellen

Das Konzept des Fundaments bietet den Vorteil, daß eine saubere Trennung zwischen der Dienstleistung der Komponente (die Simulationsaufgabe) und den sogenannten *Metainformationen* möglich wird (Metainformationen sind Informationen,

die über die eigentliche Komponentenimplementation hinausgehen, beispielsweise welche Laufzeitumgebung die Komponente erwartet; vergleiche Kapitel 5.2.2). Dadurch kann der unerwünschte Seiteneffekt vermieden werden, daß eine Komponente ansonsten nur im Zusammenhang mit der verwaltenden Infrastruktur verwendbar ist. Übertragen auf das Modellmanagement heißt das, für die Prozesse macht es kein Unterschied, ob *NeMO* zum Einsatz kommt oder nicht, was als eine der zentralen Anforderungen an *Nemo's Model Organizer* formuliert worden ist.

Das Fundament kann *NeMO* bereitstellen, indem es sich Abbilder der Prozesse und somit auch der Prozeßstruktur schafft (siehe Abbildung 6.7). Somit existiert für jeden Prozeß in der Simulation ein Ebenbild, welches *NeMO* als Schablone dient, um einem Prozeß das Fundament zur Verfügung zu stellen, welches seiner „Form" entspricht. Die Abbilder der Prozesse sind genauso wie die „Originale" (die Prozesse selbst) transient, d.h. sie existieren nur zur Laufzeit.

Eine Abbildung eines Simulationsprogrammes hingegen kann dauerhaft festgehalten werden. Mit ihrer Hilfe kann das Ebenbild eines Prozesses generiert werden, analog zu den realen Gegebenheiten, wo ein Simulationsprozeß aus einem Simulationsprogramm hervorgeht. Die Programmabbilder sind unter anderem Bestandteil von *NeMO's Simulation Database* (NeSD), welche in Abschnitt 7.2 eingehender vorgestellt wird.

6.3.2 Details

Simulationsprozeß

Die direkten Eingriffs- oder Interaktionsmöglichkeiten hinsichtlich eines Simulationsprozesses sind wegen der Vermeidung von Seiteneffekten (s.o.) auf wenige standardisierte Formen beschränkt (siehe Abbildung 6.8).

Dazu gehört auf der einen Seite eine grundlegende Form der Ausführungskontrolle (Starten und Beenden eines Prozesses) sowie auf der anderen Seite ein Ausgang zum bzw. ein Eingang vom Datenmanagement. Die Interaktion mit dem Datenmanagement kann **nicht** von *Nemo's Model Organizer* **kontrolliert**, aber zumindest **beobachtet** werden (Stichwort *Octopus System Interface*; siehe [Eng01]). Anhand einer Analyse der ausgetauschten Daten eines Prozesses wird dieser dann in den Gesamtkontext der Simulation eingeordnet.

Das Datenmanagement stellt sicherlich keinen allgemeingültigen Standard zur Inter-Prozeß-Kommunikation dar. Jedoch im Rahmen der verteilten Simulationsarchitek-

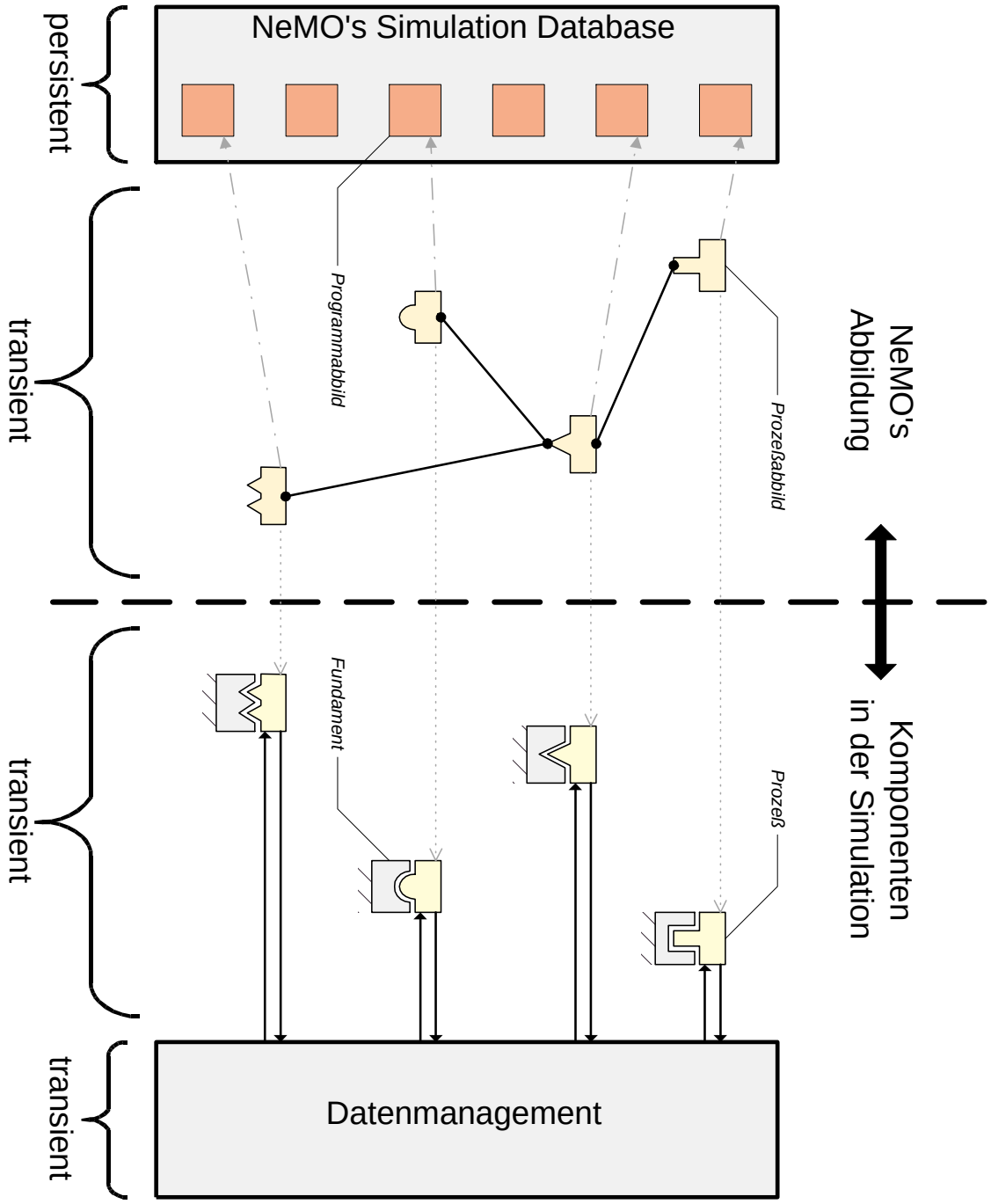


Abb. 6.7: *NeMO* als Beobachter der Simulation

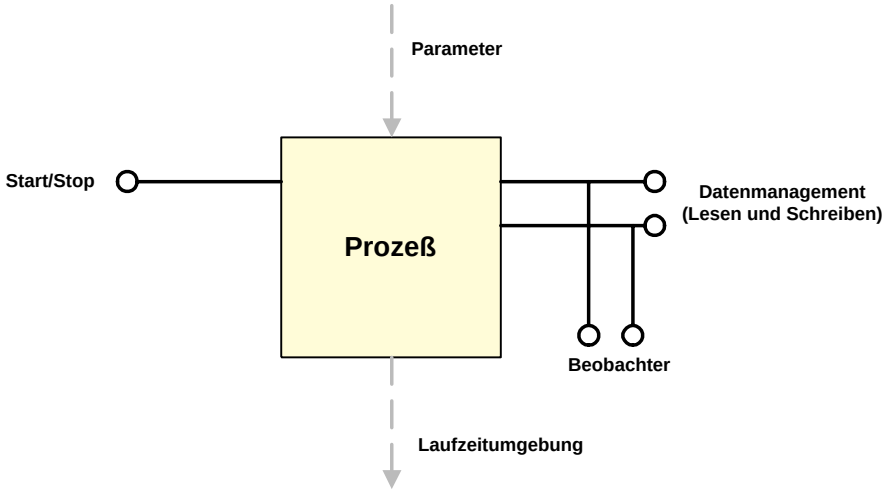


Abb. 6.8: *NeMO*'s Komponentenmodell - Prozeß

tur *DSPA* ist die Schnittstelle zum Datenpool die einzig zugelassene Methode, Informationen von einem Prozeß zu einem anderen zu übertragen. Demzufolge handelt es sich hierbei um einen lokalen oder Quasi-Standard, dem jede Komponente entsprechen muß.

Neben den direkten Formen gibt es noch zwei indirekte Möglichkeiten, wie *NeMO* auf einen Prozeß einwirken bzw. wie dieser auf *NeMO* rückwirken kann. Erstere besteht darin, dem Prozeß zur Startzeit Parameter zu übergeben, die er entsprechend seiner individuellen Logik interpretiert und sein Verhalten danach ausrichtet. Dazu wird auf die Mechanismen des Betriebssystems zurückgegriffen, was aber in diesem Falle keine Einschränkung darstellt, weil alle gängigen Betriebssysteme die Parameterübergabe an Prozesse auf nahezu einheitliche Art und Weise unterstützen.

Über den „Umweg“ der Abbildung eines realen Prozesses (siehe Abbildung 6.7) teilt dieser dem Modellmanagement mit, wie seine ganz individuelle Laufzeitumgebung aussehen soll, so daß er darauf aufbauend seine Simulationsaufgabe wie gewünscht erfüllen kann. *Nemo's Model Organizer* überprüft, ob er diese Umgebung bereitstellen kann und ob die benötigten Hardware-Ressourcen mit der entsprechenden Leistungsfähigkeit vorhanden sind (Stichwort Vorverwaltung; vergleiche Abschnitt 6.1). Ist das der Fall, kann der Prozeß ausgeführt werden, ansonsten nicht.

Prozeßabbild

Das Prozeßabbild (vergleiche Abbildung 6.9) ist eher als passiver Datenspeicher zu verstehen, und nicht als agierendes Element. Es hält den Zustand des realen Prozesses aus der Sicht des Modellmanagements fest, so daß *NeMO* auf die oben beschriebene Art und Weise mit dem Simulationsprozeß interagieren kann.

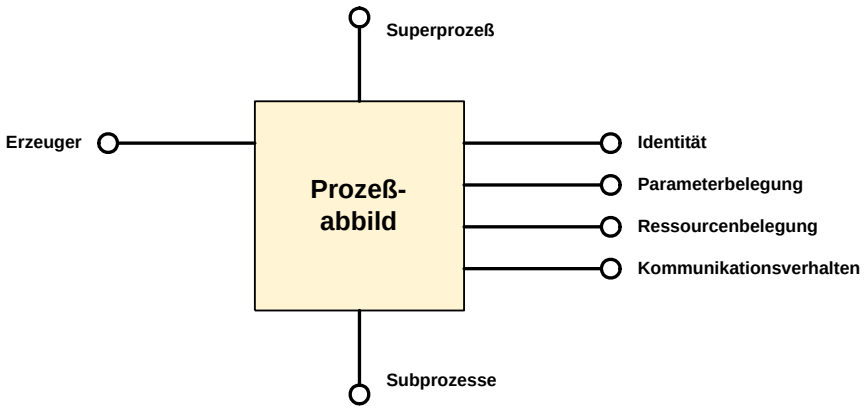


Abb. 6.9: *NeMO*'s Komponentenmodell - Prozeßabbild

Der Zustand eines Prozesses wird anhand von *Eigenschaften* und *Verknüpfungen* beschrieben, welche über Schnittstellen des Prozeßabbilds zur Verfügung gestellt werden.

- Eigenschaften

  1. *Identität*

     Mit Hilfe von Informationen über die Identität eines Prozesses kann dieser eindeutig bestimmt und lokalisiert werden. Unter diese Kategorie fallen die Bezeichnung des Prozesses (*Name*), das Verzeichnis (*Path*) und der Name (*file*) der ausführbaren Datei, aus der der Prozeß hervorgegangen ist. Weiterhin speichert das Prozeßabbild den Rechnernamen (*Host*) bzw. die Rechneridentifikation (*IP-Address*).

     Zusätzlich zu den Informationen zur reinen Unterscheidung des Prozesses können mit Hilfe des Abbildes die Befehlszeile zur Ausführung des Prozesses (*Command-Line*), die textbasierte Ausgabe des Prozesses (*Shell-Output*) sowie die Umgebungsvariablen des Prozesses (*Environment*; nutzerdefinierte Liste von Name-Wert-Paaren, die das Betriebssystem dem Prozeß übermittelt) erfragt werden.

2. *Parameterbelegung*

   Über diese Schnittstelle wird die Liste mit den Parametern des Prozesses - Name und Wert - mitgeteilt. Dabei wird unterschieden nach Art und Status des Parameters. Ersteres differenziert danach, ob der Parameter über die Kommandozeile oder über die Prozeßumgebung (*Environment*) übergeben wird, letzteres ob der Parameter persistent oder transient ist. Diese Information wird benötigt, wenn die aktuelle Simulationskonfiguration abgespeichert werden soll.

3. *Ressourcenbelegung*

   Die Ressourcenbelegung gibt an, welche Anforderungen der Prozeß hinsichtlich Rechnerkern und Ein-/Ausgabegeräten (Kapitel 6.1.2) stellt und welche Bausteine (Kapitel 6.1.1) dem Prozeß in Entsprechung der Anforderungen vom Modellmanagement zugeteilt worden sind.

4. *Kommunikationsverhalten*

   Nemo's Model Organizer erfährt vom Datenmanagement, welche Aliase der Prozeß zum Lesen und Schreiben anmeldet (siehe 6.2). Diese werden im Abbild festgehalten und können über diese Schnittstelle jederzeit wieder abgefragt werden.

- Verknüpfungen

  1. *Erzeuger*

     Das Prozeßabbild ist aus einem Programmabbild hervorgegangen. Mit Hilfe diese Schnittstelle kann man zu dem entsprechendem Programmabbild gelangen.

  2. *Prozeßstruktur*

     Die Prozesse können inhaltlich gesehen zu Prozeßgruppen zusammengefaßt werden. Beispielsweise ist es sinnvoll, die Prozesse, die die Starrkörperdynamik eines Flugzeuges, die Sensoren eines Flugzeuges und das automatische Regelsystem des Flugzeugs wiedergeben, als eine Gruppe von Prozessen zu betrachten, die gemeinsam zur Ausführung kommen sollten.

     Eine derartige Gruppierung kann durch das Modellmanagement in Form von Hauptprozessen und Subprozessen festgehalten werden, zwischen denen mit Hilfe dieser Schnittstelle navigiert werden kann.

Eine weitere Möglichkeit, sich in der von den Prozessen aufgespannten Struktur zu bewegen, ist die Verfolgung der Aliase, die sie verbinden (Betrachtung der Datenflüsse, vergleiche Abschnitt 4.3). Dazu langt das Prozeßabbild allein jedoch nicht aus, sondern es wird auch ein Abbild des Datenpools benötigt, welcher aus einer übergeordneten Sicht heraus die in der Simulation ausgetauschten Aliase wiedergibt. Ein solches Abbild des Datenpools ist ebenfalls Bestandteil von *Nemo's Model Organizer*.

Programmabbild

Ähnlich wie die Abbildung des Prozesses läßt sich das Programmabbild als Objekt zur Wissensspeicherung charakterisieren, wobei das Wissen dem Modellmanagement über definierte Schnittstellen zur Verfügung gestellt wird (siehe Abbildung 6.10). Der Begriff „Programmabbild" ist hierbei in einem weiterem Sinne als in Kapitel 6.2.3 zu verstehen, wo er nur im Zusammenhang mit Aliasen und dem Kommunikationsverhalten von Prozessen gebraucht wurde.

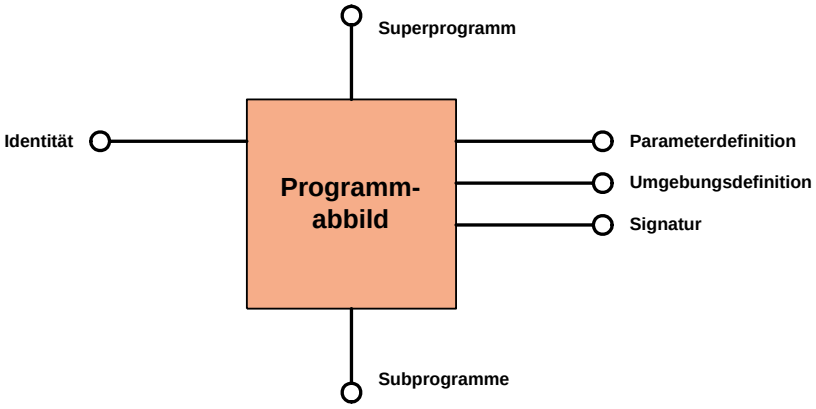


Abb. 6.10: *NeMO*'s Komponentenmodell - Programmabbild

Die Unterteilung der Schnittstellen ist analog zum Abbild eines Prozesses:

- Eigenschaften

  1. *Identität*

     Unter der Identität sind zuallererst die Informationen zu verstehen, die dazu dienen, daß Programm in *NeMO's Simulation Database* (NeSD) wiederzufinden, wie beispielsweise Verzeichnis und Name der ausführbaren Datei (siehe Kapitel 7.2.2).

Vergleichbar zusätzlicher Attribute von Dateien wird ein Simulationsprogramm durch weitere Informationen spezifiziert. Dazu gehören der Programm-Typ, eine kurze Programm-Dokumentation sowie eine Gebrauchsanweisung („Usage"-Information). Unter Typ ist hierbei die Festlegung zu verstehen, ob es sich um ein Ausführungs- oder ein Kontrollpart handelt (siehe Abschnitte 3.3 und 6.3.3).

Von wesentlicher Bedeutung für das Modellmanagement ist die Verwaltung verschiedener Versionen von Programmen. Die dazu benötigten Informationen werden ebenfalls durch das Programmabbild zur Verfügung gestellt.

2. *Parameterdefinition*

   Aus der Sicht von *NeMO* beinhaltet die Definition von Parametern eines Programms mehr als nur die Festlegung von Anzahl und Namen. Deswegen muß für jeden Parameter spezifiziert werden, ob er über die Befehlszeile oder über das Environment dem Prozeß übermittelt wird. Davon hängt ab, ob eine Information über die Plazierung des Parameters in der Befehlszeile oder ein Name für die Umgebungsvariable benötigt wird.

   Von der Parameterbelegung kann ebenfalls die vom Prozeß erwartete Laufzeitumgebung als auch das Kommunikationsverhalten beeinflußt werden. Ist dies der Fall, so müssen die Parameter entsprechend gekennzeichnet werden.

   Zur Unterstützung eines späteren Benutzers der Simulation können für jeden Parameter mögliche Wertebelegungen, Voreinstellungen und eine kurze Beschreibung desselben registriert und über das Programmabbild abgerufen werden.

3. *Umgebungsdefinition*

   Unter die Definition der Umgebung fallen drei Bereiche: die Anforderungen an Ressourcen, welche Betriebssysteme werden von einem Programm unterstützt und welche gemeinsam genutzten Bibliotheken (sogenannte *„Shared Objects"* oder *„Dynamic Link Libraries"*) erwartet der Prozeß zur Ausführung.

   Die Hardware-Anforderungen zur Ausführung eines Prozesses sind ausführlich in Kapitel 6.1 beschrieben worden. Hier sei nur ergänzend angemerkt, daß die Anforderungen von der Anzahl und Belegung der Pa-

rameter eines Prozesses abhängen können. Dieser Umstand wird auch vom Programmabbild berücksichtigt.

Die Beschreibung der unterstützen Betriebssysteme und der benötigten Bibliotheken umfaßt im wesentlichen einen Namen und eine Versionsinformation. In beiden Fällen muß *Nemo's Model Organizer* wissen, wie und wo die Dateien (das „Executable" oder die „Dynamic Link Library") in *NeMO's Simulation Database* (NeSD) (siehe unten, Abschnitt 7.2.2) zu finden sind.

4. *Programmsignatur*

   Unter Programmsignatur ist die in Abbildung 6.5 dargestellte, abstrakte Modellierung der Funktion eines Programms im Hinblick auf das zu erwartende Kommunikationsverhalten von Prozessen zu verstehen.

   Die vom Abbild des Programms bereitgestellten parametrisierten Aliaslisten können in Abhängigkeit von Anzahl und Werten der Programmparameter variieren.

- Verknüpfungen

  1. *Programmstruktur*

     Die für die Prozeßabbilder beschriebene Möglichkeit der Gruppierung wird erreicht durch eine analoge Gruppierung der Programmabbilder in Hauptprogramme und deren Subprogramme.

     Eine Strukturierung der Programme anhand der für jedes Programm abgespeicherten, parametrisierten Aliaslisten wäre theoretisch denkbar. Der Aufwand für eine generelle und unbedingte Betrachtung aller möglichen Datenflußkombinationen ist jedoch unverhältnismäßig hoch und bietet auch keinen unmittelbar erkennbaren Nutzen, so daß davon abgesehen worden ist.

6.3.3 Richtlinien

Die im Zusammenhang mit *NeMO*'s Komponentenmodell formulierten Richtlinien sollen für den Entwickler einer Komponente in der verteilten Simulationsarchitektur *DSPA* Hinweise geben, unter welchen Umständen das Modellmanagement seine optimale Leistungsfähigkeit entfalten kann.

Eine Nichtbeachtung der Richtlinien führt aber nicht gleich zu einem Totalausfall von *Nemo's Model Organizer* oder dazu, daß eine Komponente unbedingt und ausnahmslos aus der Simulationsarchitektur entfernt wird, sondern muß von Fall zu Fall vom Administrator der Simulationsarchitektur bewertet werden. Eine Veränderung eines Programmes wird in keinem Falle vom Modellmanagement vorgenommen werden, sondern im Bedarfsfall vom entsprechenden Entwickler.

Die Richtlinien können hinsichtlich ihrer Zielrichtung in zwei Bereiche unterteilt werden.

Kommunikation

Grundlegende Voraussetzung für alle Komponenten in der *DSPA* ist die alleinige Verwendung der Schnittstelle des Datenmanagements zum Zwecke des Datenaustauschs. Andere Kommunikationsformen können von *NeMO* nicht beobachtet und damit auch nicht berücksichtigt werden. Der Extremfall, daß ein Prozeß nur über andere Schnittstellen kommuniziert, führt dazu, daß er lediglich wie ein Systemprozeß behandelt werden kann (siehe Abschnitt 3.3), sprich das Modellmanagement verwaltet einzig seine Laufzeitumgebung.

In Ergänzung zur Definition durch das Datenmanagement wird eine Konvention für die Benamung von Aliasen festgelegt. Diese wurde schon im Detail im Kapitel 6.2.2 vorgestellt. Sie dient der Strukturierung der in einer Simulation ausgetauschten Daten, welche dadurch durch den Menschen besser und intuitiver erfaßt werden können.

Die zusätzlich formulierten Regeln für die parametrisierten Aliase sind wesentlich für die Analyse des Kommunikationsverhaltens eines Programms bzw. eines Prozesses. Die Werte für die Parameter werden entweder über die Kommandozeile oder über das Environment (s.o.) übergeben und müssen **unverändert** in die Aliase eingehen. Eine Veränderung der Werte im Inneren des Prozesses kann von *NeMO* **nicht registriert** werden.

Die Verletzung der Namenskonvention oder der Regeln kann zu Fehlinterpretationen des Kommunikationsverhaltens seitens des Modellmanagements führen, wodurch die Einordnung eines einzelnen Programms oder Prozesses in den Kontext des Gesamtmodells beeinträchtigt wird.

Abschließend ist jeder Entwickler angehalten, alle Aliase, die von seinem Simulationsmodul geschrieben werden, über eine unabhängige und zentrale Datenbank zu

registrieren und zu veröffentlichen (vergleiche Abschnitt 7.2.3). Damit existiert eine eindeutige Anlaufstelle, wo die in einem Simulationsmodell produzierten Informationen „offline“ begutachtet und analysiert werden können.

Ablaufsteuerung

Für eine optimale durch das Modellmanagement koordinierte Ablaufsteuerung der Simulation wird eine Trennung von Kontroll- und Ausführungspart eines Simulationsmoduls vorgeschrieben (siehe Abbildung 6.11).

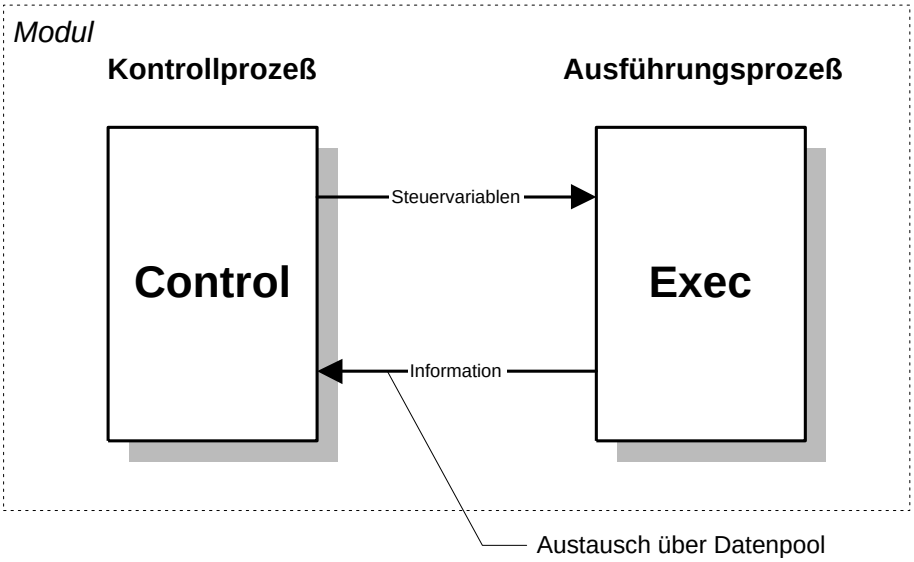


Abb. 6.11: Trennung von Kontrolle und Ausführung

Der Kontrollprozeß erfüllt keine inhaltlichen Aufgaben hinsichtlich des Simulationsmodells, sondern kontrolliert über Steuervariablen das eigentliche Simulationsprogramm. Dazu kann er auf Informationen vom Ausführungsprozeß zurückgreifen. Der Datenaustausch zwischen den beiden Teile eines Simulationsmoduls findet ebenfalls **nur** über das Datenmanagement statt.

Damit wird eine saubere Trennung zwischen Modellumsetzung und Eingriffsmöglichkeiten durch den Menschen erreicht, so daß *NeMO* alle Schnittstellen zur Kontrolle der Simulationsprozesse koordiniert über einen zentralen Arbeitsplatz bereitstellen kann.

Zum minimalen Leistungsumfang eines Kontrollprozesses gehört eine rudimentäre Ablaufsteuerung:

- *Run*

  Normaler Prozeßablauf im Zuge einer Simulation

- *Stop*

  Der Prozeß verbleibt in seinem gegenwärtigen Zustand. Dies entspricht einer Pause in der Simulation.

- *Reset*

  Existiert für einen Prozeß ein definierter Anfangszustand, wie beispielsweise die Startposition eines Flugzeugs, dann kann er mit Hilfe dieser Anweisung dorthin zurückgesetzt werden.

- *Exit*

  Dadurch wird dem Prozeß ein kontrollierter Abbruch signalisiert.

Alle Kontrollprozesse können analog zu den Ausführungsprozessen in eine Hierarchie eingeordnet werden, welche sich von der globalen Simulations- zur gezielten Prozeßsteuerung hin orientiert (vergleiche Abbildung 6.12).

Es gilt das Prinzip, daß übergeordnete Controller Priorität besitzen und daß Befehle von oben nach unten weitergegeben werden, d.h. wenn der *Group Controller Flugzeuge* ein *exit*-Befehl sendet, so müssen alle untergeordneten *Group Controller* diesen an die ihnen zugeordneten *Process Controller* weitergeben, welche wiederum die spezifischen Ausführungsprozesse benachrichtigen. Befehle des *World Controllers* (oberster Kontrollprozeß, in *NeMO* integriert) müssen von jedem Prozeß unabhängig von dessen Position in der Hierarchie erfragt und befolgt werden.

Damit ergibt sich für jedes Simulationsmodul die in Abbildung 6.13 dargestellte Befehlsstruktur.

Kontrollprozesse werden in der Regel dem Benutzer eine graphische Bedienoberfläche zur Steuerung der Prozesse bieten. Um eine einheitliche Bedienung und ein einheitliches Aussehen („look-and-feel“) der Oberflächen zu gewährleisten, wird an dieser Stelle ein Vorschlag zur Gestaltung der *Graphical User Interfaces* (GUI) gemacht (siehe Abbildung 6.14).

Die Bedienoberfläche sollte möglichst nur aus einem Fenster bestehen, es sei denn es handelt sich um ein Modul mit vielen Steuervariablen. Hintergrund ist die Tatsache, daß ein Benutzer mit den Fenstern aller Bedienoberflächen konfrontiert wird,

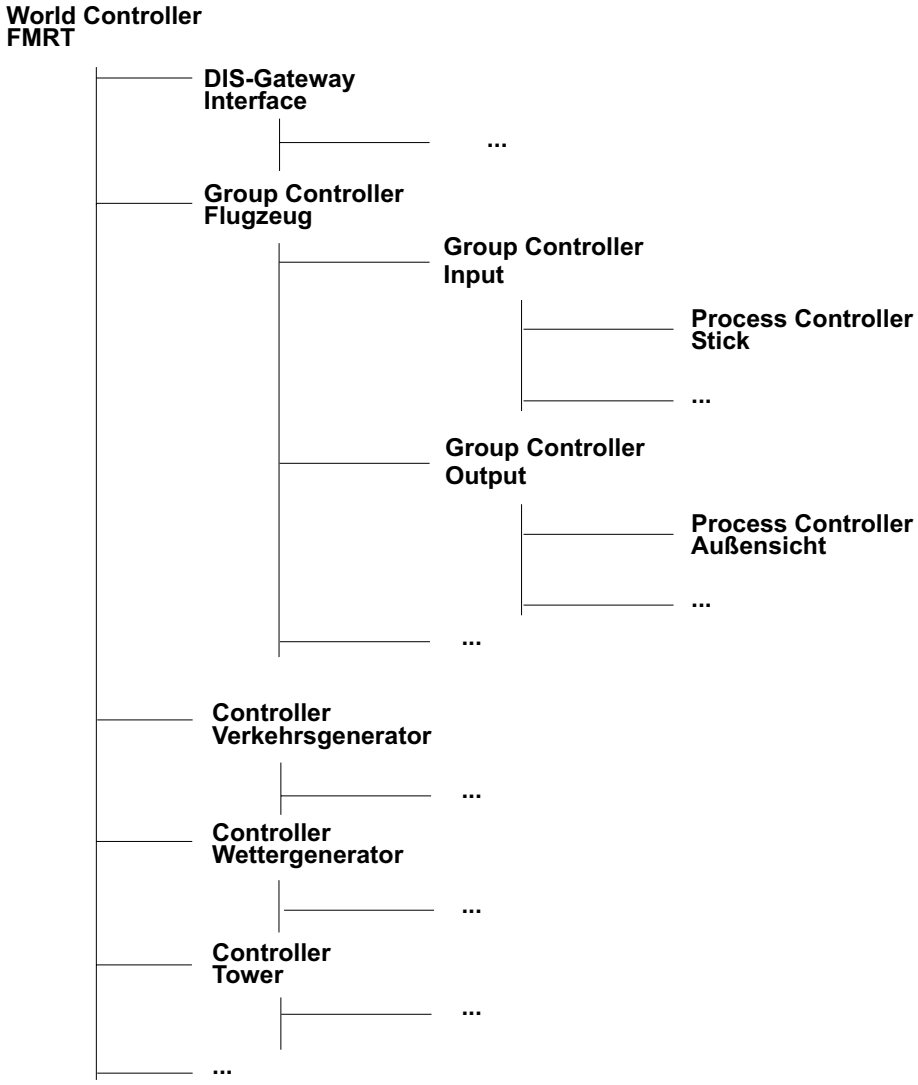


Abb. 6.12: Kontrollhierarchie

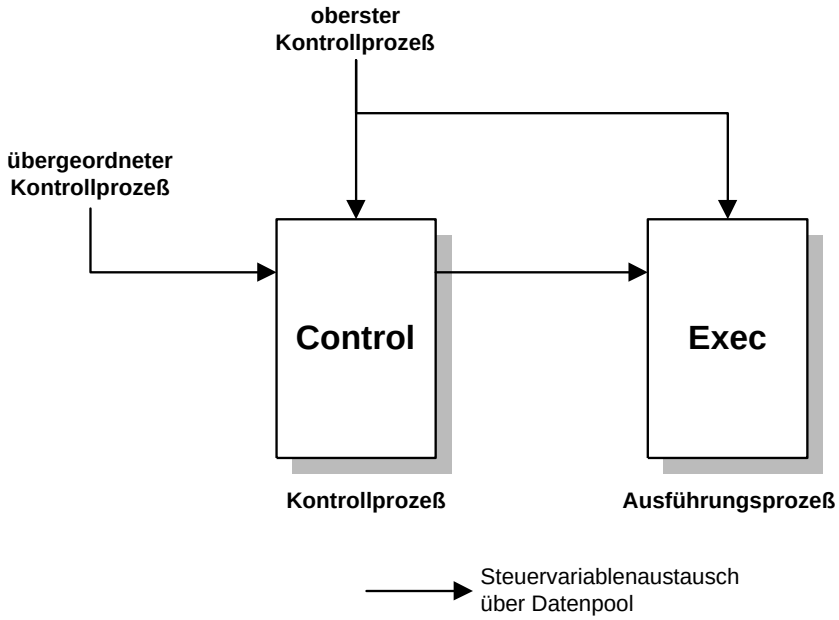


Abb. 6.13: Gesamte Befehlsstruktur eines Simulationsmoduls

Abb. 6.14: Kontroll-Oberflächen-Layout

so daß die Gesamtanzahl möglichst gering zu halten ist. Alle Bezeichnungen sind konsequent in englischer Sprache durchzuführen.

Auf dem Rahmen der Oberfläche wird der Name des Kontrollprozesses und der Name der Simulationswelt angezeigt. Ersterer besteht aus einem allgemeinen Teil plus den zugeordneten Parametern und letzteres ist ein globaler Bezeichner für die Simulation (siehe [Eng01]).

Allgemein	:	<WORLD> : <NAME>
Beispiel	:	WORLD1 : ControlCollision_ACR1

Am Kopfende des Fensters befindet sich eine Menü-Leiste, in der linksbündig das Hauptmenü zu finden ist. Dieses enthält zumindest den Eintrag *Exit*, worüber der Kontrollprozeß beendet werden kann. Rechtsbündig findet sich ein Kürzel des verantwortlichen Entwicklers sowie eine Versionsinformation in Form von <*Monat/Jahr*>.

Unterhalb der Menüleiste ist ein frei zu gestaltender Bereich („user-defined-area“). Hierunter ist der prozeßspezifische Teil der Kontrolle (Aktionen und Rückmeldung) zu verstehen, welcher von seiner Aufgabe in der Simulation abhängt.

In dem darunterliegenden Feld wird die grundlegende Ablaufsteuerung (*Run*, *Stop*, *Reset*, *Exit*; siehe oben) integriert. Dazu werden sogenannte *Radio-Buttons* eingesetzt, die sich dadurch auszeichnen, daß immer nur einer aus der Gruppe aktiviert sein kann.

Am unteren Ende des Fensters kann der Übersicht wegen noch der Name der Einheit, die der Kontroll- und der Ausführungsprozeß bilden, angezeigt werden. Eine beispielhafte Umsetzung der Gestaltungsrichtlinien ist in Abbildung 6.15 zu erkennen.

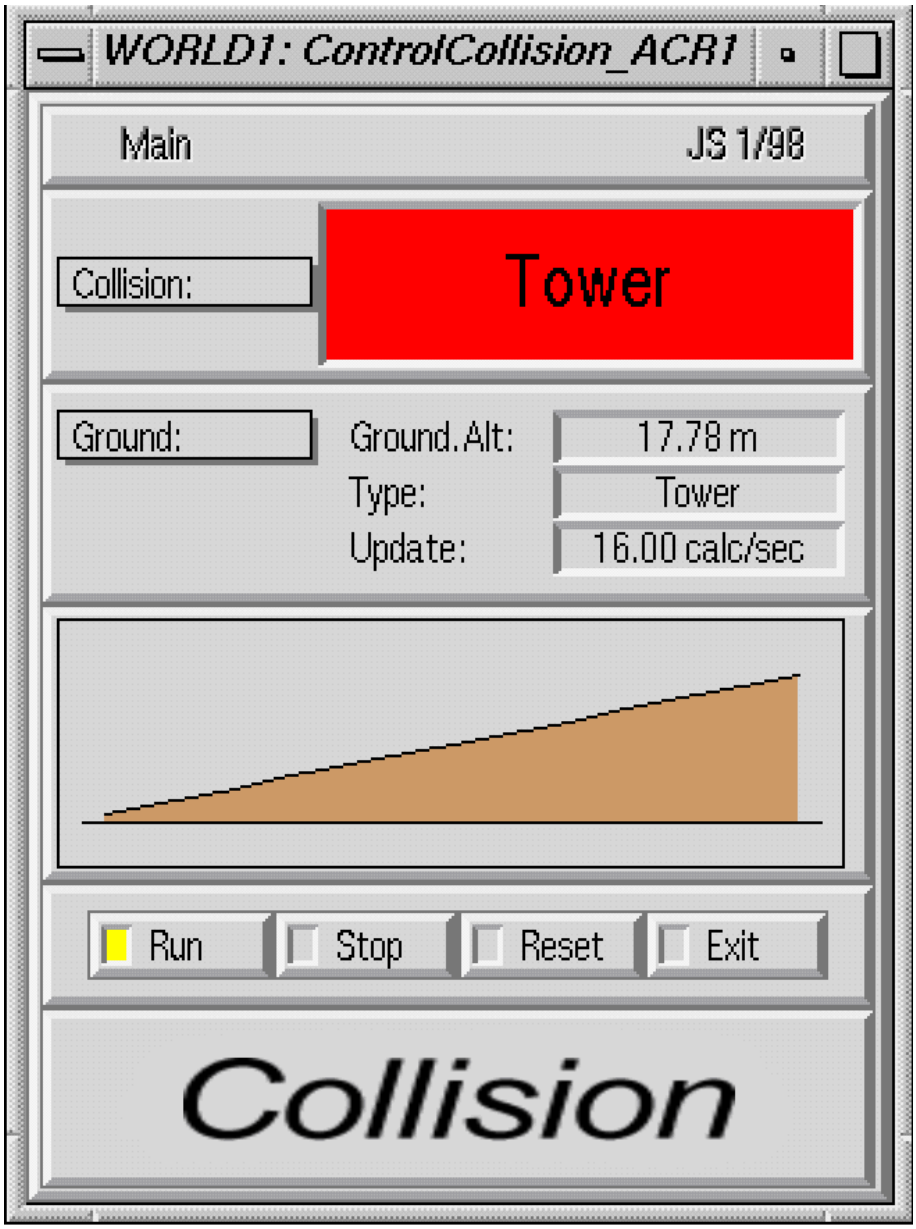


Abb. 6.15: Beispiel zur Gestaltung einer Kontrolloberfläche

6.3.4 Eincheckvorgang

Der Eincheckvorgang ist eine Anleitung für Entwickler und Administratoren, wie eine neue Komponente unter Zuhilfenahme des Modellmanagements in die verteilte Simulationsarchitektur *DSPA* zu integrieren ist. Die Bedeutung des Vorgangs liegt darin begründet, daß dieser die **einzige** Informationsquelle für *Nemo's Model Organizer* außer dem Datenmanagement darstellt. Demzufolge sollte er gewissenhaft durchgeführt werden, um das oben vorgestellte, persistente Programmabbild mit allen benötigten Informationen zu versorgen.

Im Zusammenhang mit der Integration von Simulationsmodulen ergeben sich drei Variationen:

- Neues Simulationsprogramm

  Alle Angaben müssen vollständig neu erfragt werden.

- Neue Version eines bestehenden Programms

  Auf bereits existierende Versionen kann als Ausgangspunkt zurückgegriffen werden. Es wird der Tatsache Rechnung getragen, daß gegenüber älteren Versionen nicht alles geändert worden sein wird.

- Änderung einer existierenden Programmversion

  Es muß nur das geändert oder ergänzt werden, was fehlerhaft oder unzureichend definiert worden ist.

Der folgende Sequenz dokumentiert die Reihenfolge der Informationsgewinnung. Dabei wird darauf eingegangen, wer die Quelle der Information ist, wie sie eventuell bearbeitet werden muß und an welcher Stelle sie dann festgehalten wird. Der hier dargestellte Ablauf bezieht sich im wesentlichen auf den Fall, daß ein neues Programm integriert werden soll. Je nach Variation (s.o.) kann es also sein, daß Teile des Eincheckvorgangs übersprungen werden.

1. Identität

   Zu Beginn des „Eincheckens“ werden vom Entwickler ein Programmbezeichner, der Name der ausführbaren Datei, der Programmtyp (Ausführungspart, Kontrollpart oder beides integriert), eine kurze Beschreibung des Programms,

eine Option zur Abfrage einer Gebrauchsanweisung („Usage“) sowie eine Information über die Kompatibilität zu vorherigen Versionen eingegeben. Diese Daten werden eins zu eins in das Programmabbild übernommen und in *NeMO*'s Datenbank abgespeichert.

2. Parameterdefinitionen

   Für die Definition der Anzahl und Art der Programmparameter kann der Entwickler auf existierende Parametertypen aus der Datenbank des Modellmanagements zurückgreifen, da bestimmte Parameter von verschiedenen Programmen benötigt werden. Handelt es sich um individuelle Parameter können diese anhand einer Eingabemaske neu definiert werden.

   Der Umfang der Information ist gleich der Beschreibung im Zusammenhang mit dem Programmabbild (vergleiche Kapitel 6.3.2).

3. Umgebung

   Die Umgebung wird beschrieben durch eine Liste von unterstützten Betriebssystemen, eine Liste der benötigten Bibliotheken („Dynamic Link Libraries“) und über die Anforderungen des Programms hinsichtlich Hardware-Ressourcen. Für eine detaillierte Beschreibung dieser Aspekte wird auf die vorangegangenen Abschnitte verwiesen.

   Zur Unterstützung des Entwicklers bei der Eingabe dieser Daten stellt das Modellmanagement entsprechende Typbeschreibungen für alle drei Kategorien zur Verfügung, anhand derer der Entwickler nur noch auszuwählen braucht. Hier wird der Tatsache Rechnung getragen, daß unterschiedliche Programme hinsichtlich ihrer Umgebung durchaus ähnliche Anforderungen stellen können. Damit diese nicht jedesmal neu definiert werden müssen, wird eine Sammlung von entsprechenden Typdefinitionen in die Datenbank des Modellmanagements integriert.

4. Subprogramme

   An dieser Stelle legt der Entwickler fest, welche anderen Programme mit seinem Modul zusammen gestartet werden sollen. Das Modellmanagement bietet ihm dazu alle bereits integrierten Programme zur Auswahl, unterschieden nach ihren jeweiligen Versionen.

5. Prozeßplazierung

   Mit dem Beginn der Prozeßplazierung ist der erste Teil der Informationsgewin-

nung abgeschlossen. Nun muß der Entwickler anhand der zur Verfügung stehenden Rechner entscheiden, auf welchem das neue Programm zur Ausführung gebracht werden soll. Dabei wird vom Modellmanagement mit Hilfe der soeben erfolgten Angaben überprüft, welche Rechner überhaupt in der Lage sind, die gewünschte Laufzeitumgebung bereitzustellen.

Sind alle Voraussetzungen erfüllt, so wird durch *NeMO* alles Notwendige zur Erzeugung eines Prozesses vorbereitet.

6. Aufzeichnung von Testläufen

   Für die Feststellung des Kommunikationsverhaltens des Programms müssen zwei Testläufe durchgeführt werden. Dazu wird der Entwickler vom Modellmanagement unter der Zuhilfenahme der zuvor formulierten Definitionen aufgefordert, die Belegung der Programmparameter vorzunehmen, wobei *NeMO* die Einhaltung der Forderung nach Eindeutigkeit aller Parameterwerte (siehe Abschnitt 6.3.3) überwacht.

   Die Ausführung des Prozesses ist sodann eine unmittelbare Überprüfung, ob alle bis dahin gemachten Angaben für eine Verwendung durch das Modellmanagement geeignet sind. Zu dem Zeitpunkt, wo der Entwickler sich davon überzeugt hat, daß der Prozeß ordnungsgemäß zu arbeiten scheint, wird mit Hilfe der Schnittstelle zum Datenmanagement eine Momentaufnahme seines Kommunikationsverhaltens gemacht

   Der zweite Testlauf wird auf die gleiche Art und Weise ausgeführt.

7. Programmsignatur

   Das Ergebnis der Testläufe sind Aufzeichnungen der Aliase, die von den Prozessen zum Lesen und Schreiben angemeldet worden sind sowie die dazugehörigen Parameterbelegungen. Mit Hilfe dieser Informationen bestimmt *NeMO* nun das Kommunikationsverhalten des Programms (Stichwort parametrisierte Aliaslisten; vergleiche Kapitel 6.2.3).

   Das Resultat der Analyse kann vom Entwickler begutachtet und gegebenenfalls manuell überarbeitet werden.

8. Abschluß

   Der Abschluß besteht in der Übernahme des Programms und der Informationen für das Programmabbild in *NeMO's Simulation Database* (NeSD) (siehe

unten). Ersteres wird vom Administrator nach den Anweisungen des Modellmanagements umgesetzt, letzteres geschieht automatisch durch *Nemo's Model Organizer*.

6.4 Konstruktion des Rahmenwerks

Nemo's Model Organizer fungiert als Rahmen um die Simulationsmodule und das Datenmanagement (vergleiche Abbildung 2.5), der einen Benutzer in die Lage versetzt, die komplexen Beziehungen und Abläufe von einem zentralen Arbeitsplatz aus zu kontrollieren. In diesem Abschnitt soll nun der interne Aufbau des Modellmanagements vorgestellt werden, da derartige Kenntnisse für die Inbetriebnahme und Wartung von *NeMO* vonnöten sind.

Die interne Konstruktion des Rahmenwerks folgt ebenfalls den Richtlinien der verteilten Simulationsarchitektur *DSPA* (Stichworte: modular, verteilt, flexibel und skalierbar; siehe Kapitel 3.3). Die Gesamtaufgabe wird in Teilaufgaben untergliedert, die von eigenständigen Objekten übernommen werden (Objekt-Orientierung). Objekte werden repräsentiert durch ihre charakteristischen Informationen und Operationen, die auf die Informationen angewendet werden können. Aus der Synthese der Einzelaufgaben zur Gesamtaufgabe von *Nemo's Model Organizer* ergibt sich die Notwendigkeit zur Verwaltung von einer Menge von Objekten und deren Beziehungen.

Der in diesem Zusammenhang benötigten abstrakten Funktionalitäten wie

- Dynamische Konstruktion und Destruktion,
- Einfügen und Entfernen von Objekten,
- Navigieren zwischen den Objekten,
- Suchen von Objekten,
- Aufspalten in bzw. Zusammenfügen von mehreren Teilstrukturen

wird am besten durch die Verwendung einer allgemeinen Baumstruktur entsprochen. Deswegen werden alle Objekte des Modellmanagements in Bäumen organisiert.

Innerhalb der gesamten Baumstruktur lassen sich Substrukturen identifizieren, die aus einer abstrahierten Sicht heraus immer wieder im Baum aufzufinden sind. Diese

Teilstrukturen stehen in Zusammenhang mit den Funktionalitäten, die von jedem Objekt benötigt, und deswegen von diesen unabhängig und in einer allgemeingültigen Form zur Verfügung gestellt werden. Im Anschluß an eine Beschreibung der Substruktur wird die Konstruktion des Rahmenwerks im gesamten vorgestellt.

6.4.1 Substruktur

NeMO's Komponentenmodell zeichnet sich durch eine Trennung von Inhalt und Verwaltung einer Komponente aus, d.h. der Entwickler beschäftigt sich hauptsächlich mit der Simulationsaufgabe der Komponente, während das Modellmanagement sich um die Verwaltung und die Bereitstellung einer Infrastruktur für die Komponente kümmert.

Ein ähnlicher Ansatz wird nun für die Interna von *NeMO* verfolgt. Hier ist beispielsweise das Objekt „Programmabbild“ (siehe Abschnitt 6.3.2) mit der inhaltlichen Aufgabe betraut, aus der Sicht des Modellmanagements den Zustand eines Simulationsprogramms möglichst vollständig und präzise wiederzugeben. Es ist jedoch nicht daran interessiert zu wissen, wie z.B. der Zugriff auf *NeMO's Simulation Database* (NeSD) erfolgt oder wie es in der gesamten Hierarchie des Modellmanagements eingeordnet wird.

Derartige, gemeinsam genutzte Funktionalitäten, als Grundlage und auch Notwendigkeit aller Objekte des Modellmanagements, werden über eine definierte und vorgegebene Substruktur bereitgestellt (vergleiche Abbildung 6.16).

Die Teilstruktur verknüpft drei Basis-Objekte (*Element*, *Container* und *Keeper*) und läßt sich in dieser Form im gesamten Objektbaum des Modellmanagements immer wieder finden. Sie bildet die Grundlage, auf der die eigentlichen Objekte (Kapitel 6.4.2) aufbauen, und soll nun im Detail vorgestellt werden.

Allen Basis-Objekten gemeinsam ist ein *Statusmanagement* und ein *Logging Mechanismus*, deren Beschreibung der Einfachheit halber vorangestellt wird.

- Statusmanagement

  Das Statusmanagement umfaßt einen Existenz- und Fehlerstatus. Ersterer besagt beispielsweise, ob ein Objekt aktiv oder inaktiv ist bzw. ob ein Objekt nur zur Laufzeit (transient) oder dauerhaft (persistent) existiert.

  Desweiteren kann über das Statusmanagement das Auftreten eines Fehlers individuell für jedes Objekt dokumentiert werden. Zusätzlich zur reinen Feh-

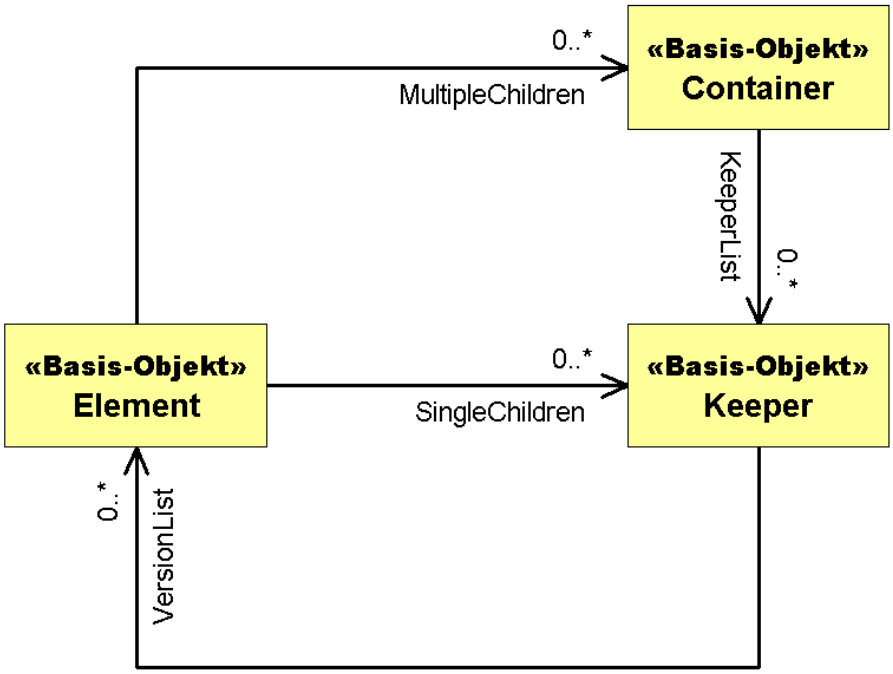


Abb. 6.16: *NeMO*'s interne Substruktur

lerbeschreibung gibt es die Möglichkeit, ergänzende Notizen und den Verlauf von Fehlern („Error-History“) festzuhalten.

- Logging Mechanismus

  Der Logging Mechanismus dient der externen Speicherung von Informationen über das Modellmanagement (Fehler, Nachrichten, ...), anhand derer der Administrator im Nachhinein das Verhalten von *NeMO* begutachten kann.

  Es wird unterschieden nach Nachrichten, die sich lokal auf einen Rechner beziehen und dann auch dort aufzufinden sind. Informationen mit allgemeiner Bedeutung werden im gesamten Rechner-Netzwerk gesammelt und an eine zentrale Stelle weitergeleitet.

Das Basis-Objekt *Element*

Von einem *Element* werden alle Objekte des Modellmanagements abgeleitet. Es erbringt folgende Dienstleistungen:

- Positionierung

  Ein jedes Objekt muß in den gesamten Objektbaum von *NeMO* unterhalb

eines anderen Knotens integriert werden. Diese Aufgabe übernimmt das Basis-Objekt *Element*. Das gleiche gilt für den umgekehrten Falle, wenn ein Objekt entfernt werden soll.

Zusätzlich stellt ein *Element* eine absolute Beschreibung der Position im Baum zur Verfügung, und zwar über eine vom Root-Element ausgehende Pfadbeschreibung. Diese wird beispielsweise bei der Speicherung von Objektbeziehungen benötigt.

Für den Administrator der Simulation kann es Situationen geben, z.B. in Fehlerfällen, wo er in Erfahrung bringen muß, wo ein Objekt ausgeführt wird (Aufhebung der Transparenz). Für diese Fälle stellt das *Element* Angaben zur Verfügung, über den Prozeß, in dem es lokalisiert ist (Rechnername und Prozeß-Id).

- Versionierung

  Über das *Element* kann bestimmt werden, ob ein Objekt versioniert sein soll oder nicht. Im ersteren Falle erzeugt das *Element* eindeutige Versionskennzeichnungen durch Bezug auf Uhrzeit und Datum der Erzeugung einer Version.

- Keepermanagement

  Über das Keepermanagement kann eine beliebige Anzahl von Keepern eines Elements koordiniert werden. Ein Keeper ist der Verwalter eines einzelnen Objekts beliebigen Typs. Er ist quasi der Aufhängungspunkt, an dem ein Kind-Element in den Baum eingehängt wird. Die indirekte Verwaltung der Kind-Elemente über einen Keeper ermöglicht unter anderem eine einfache Versionsverwaltung von Objekten (vergleiche Beschreibung des Basis-Objekts *Keeper*).

- Containermanagement

  Das Containermanagement stellt eine Erweiterung des Keepermanagements dar, für den Fall, daß ein Element mehrere Kind-Elemente des gleichen Typs besitzt. In diesem Fall werden die entsprechenden Verwalter (*Keeper*) in einem *Container* zusammengefaßt.

- Referenzmanagement

  Objekte können komplexe Beziehungen untereinander aufbauen, indem sie Verweise aufeinander besitzen. Diese müssen verwaltet werden, um einerseits

den Überblick zu behalten, wer auf wen zugreifen möchte. Andererseits ist das Referenzmanagement von sehr großer Bedeutung, wenn ein Objekt entfernt wird. In diesem Falle müssen alle Halter von Referenzen benachrichtigt werden, daß das entsprechende Objekt demnächst nicht mehr existieren wird.

- Fehlerbehandlungsmechanismus

  In Anlehnung an den *Exception*-Mechanismus in der Programmiersprache *C++* ist in jedem *Element* ein Fehlerbehandlungsmechanismus integriert. Tritt ein Fehler auf, den ein Objekt nicht selbständig beheben kann, so wird er durch diesen Mechanismus im Objektbaum solange nach oben weitergegeben, bis ein anderes Objekt in der Lage ist, entsprechende Maßnahmen zu ergreifen. Im Extremfall wird der Fehler bis zur Wurzel des Baums weitergeleitet, und von dieser an den Menschen, falls auch das Root-Objekt den Fehler nicht beseitigen kann.

  Auf diese Art und Weise muß ein Objekt zur Fehlerbehebung kein übergeordnetes Kontextwissen besitzen, denn Fehlererkennung und -behebung werden sauber getrennt, was eine der grundlegenden Ideen der *Exceptions* ist.

- Datenbankanbindung und -koordination

  Die Anbindung aller Objekte an *NeMO's Simulation Database* (NeSD) erfolgt über das Basis-Element; dieses stellt den Zugriff her und gibt auch vor, auf welche Art und Weise er zu geschehen hat.

  Ein jedes Objekt ist nur für seinen eigenen Zustand verantwortlich, d.h. es liest oder schreibt nur Informationen, die den eigenen Zustand repräsentieren. Über das Basis-Objekt *Element* wird dafür gesorgt, daß ein Lese- oder Schreibaufruf in der richtigen Reihenfolge innerhalb des Baums weitergegeben wird. Obwohl es für jedes Element dadurch so aussieht, als ob einzig und allein Daten übermittelt werden, die seinen eigenen Zustand betreffen, wird gleichzeitig die Struktur des Baums an einen externen Speicher übergeben oder von diesem erfahren.

- Environment

  Jedes *Element* bietet als eine dynamische Erweiterungsmöglichkeit zur Beschreibung seiner Eigenschaften ein sogenanntes *Environment*, welches eine Übertragung des Prinzips der Prozeß-Umgebung (s.o.) auf Objektebene darstellt.

Das Environment ist eine Liste von Name-Wert-Paaren, die beispielsweise dem Menschen zusätzliche Informationen über ein Objekt vermitteln können oder einfach an die zu verwaltenden Prozesse weitergegeben werden.

Das Basis-Objekt *Keeper*

Ein *Keeper* ist der Verwalter eines einzelnen Kind-Elements. Die indirekte Form der Einbindung von Kind-Elementen ist gewählt worden, um auf diese Art und Weise folgende Dienstleistungen zu erbringen:

- Versionsmanagement

  Ein *Keeper* ist verantwortlich für die Versionen eines einzelnen Objekts von einem bestimmten Typ. Er erzeugt und löscht Versionen, ordnet sie und bestimmt die aktuell gültige Version.

  Alle Versionen werden über den *Keeper* in den Baum integriert, der damit auch die Position der Objekte im Baum festlegt.

- Factory-Anbindung

  Ein *Keeper* ist prinzipiell für Objekte unterschiedlichen Typs verwendbar, d.h. der Objekttyp, den er verwaltet, ist nicht von vornherein festgelegt. Es muß also einen allgemeingültigen Mechanismus geben, wie Objekte von beliebigem Typ generiert werden können, und zwar mit der Hilfe sogenannter *Factory*-Objekte, die einzig für die Erzeugung eines bestimmten Objekttyps existieren.

  Aus diesem Grunde besitzt ein *Keeper* für die Erfüllung seiner Aufgaben eine Anbindung an das Factory-Objekt, welches dem Typ des verwalteten Objekts entspricht.

- Datenbankanbindung und -koordination

  Ebenso wie ein *Element* ist ein *Keeper* in die Anbindung an *NeMO's Simulation Database* (NeSD) miteinbezogen. Seine Aufgabe besteht darin, alle Informationen, die die Versionsverwaltung betreffen, in der Datenbank zu speichern bzw. im umgekehrten Falle, die in der Datenbank beschriebene Versionsstruktur wiederherzustellen.

  In die vom *Keeper* beschriebene Datenstruktur werden die versionierten Elemente eingepaßt, indem der Zugriff auf die Datenbank vom Keeper an den entsprechenden Stellen an die Elemente weitergeleitet wird.

Ein *Keeper* kann auch in einer leicht abgewandelten Form auftreten, und zwar als sogenannter *Link-Keeper*. Diese Variation eines Keepers ist gedacht für die Verwaltung von Verweisen auf Objekte, die in einem ganz anderen Bereich des Baums existieren (siehe Abbildung 6.17) und ermöglicht die gemeinsame Nutzung von Objekten.

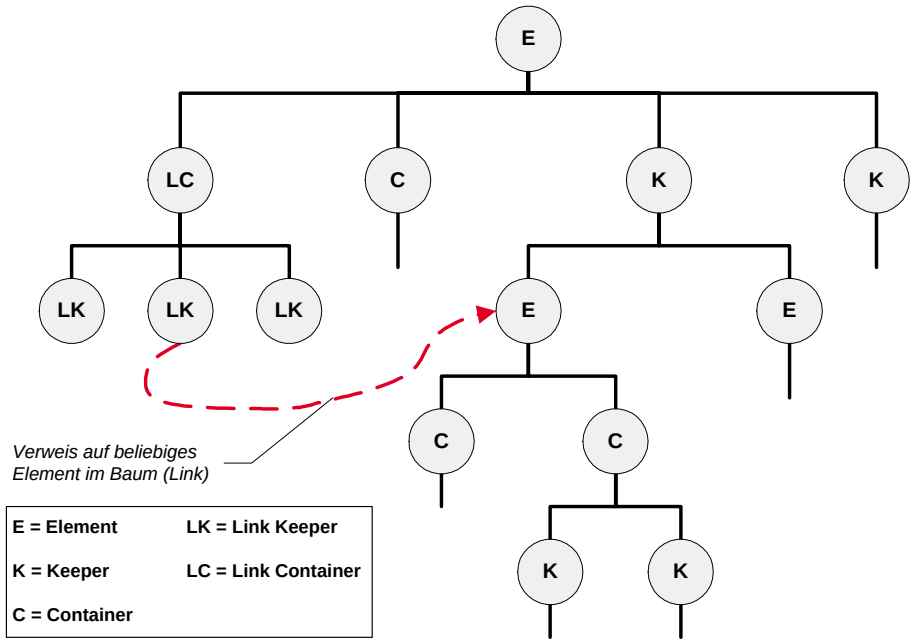


Abb. 6.17: *NeMO*'s Link Keeper

Ein *Link-Keeper* ist nicht verantwortlich für die Erzeugung und Zerstörung der Objekte, auf die er verweist. Anstelle der Anbindung an *Factory*-Objekte ist er deswegen mit einem sogenannten *Trading Service* verbunden. Über diesen lassen sich alle bekannten Objekte eines Typs auffinden, ähnlich dem Prinzip eines Branchen-Telefonbuches („Gelbe Seiten"), aus denen der *Link-Keeper* dann eines auswählen kann. Bei versionierten Objekten kann durch Angabe von Richtlinien genauer spezifiziert werden, welche der Versionen der *Link-Keeper* verwalten soll (nur die aktuelle oder die neueste, eine bestimmte Versionsnummer oder alle Versionen).

Das Basis-Objekt *Container*

Ein *Container* koordiniert eine Menge von Keepern, die alle den gleichen Objekttyp verwalten. Er entlastet dadurch das *Element*, welches ansonsten jeden *Keeper* einzeln koordinieren müßte. Die Aufgaben des *Containers* können folgendermaßen beschrieben werden:

- Keepermanagement

  Der *Container* ist verantwortlich für die Erzeugung von Keepern eines bestimmten Objekttyps und deren Verwaltung (Veröffentlichung der Identität der Keeper, Angebot von Suchfunktionalitäten, Entfernen von nicht mehr benötigten Keepern).

  Der *Container* sorgt für die Integration seiner *Keeper* in den Baum und legt damit auch deren Position fest.

- Factory-Anbindung

  Ein *Container* besitzt Kenntnisse, wie *Keeper* zu generieren sind, jedoch nicht wie Objekte beliebigen Typs erzeugt werden können. Da aber die Erzeugung eines *Keepers* meist mit einer eines Objekts einhergeht, ist der *Container* ebenso wie ein *Keeper* an den *Factory*-Mechanismus angeschlossen (siehe oben; Beschreibung des Basis Objekts *Keeper*).

- Datenbankanbindung und -koordination

  Im Hinblick auf die Speicherung bzw. das Einlesen von Daten übernimmt der *Container* ähnliche Aufgaben wie ein *Keeper*. Im ersteren Falle ist er verantwortlich für die Erzeugung einer Datenstruktur, in der die *Keeper* und *Elemente* eingebettet werden können. Beim zweiten Fall, dem Einlesen von einer Datenbank, wird die Beschreibung der Struktur von *Keepern* in eine entsprechende Objektstruktur umgesetzt und der Lese-Aufruf in der entsprechenden Reihenfolge an die *Keeper* weitergegeben.

  Zur Wahrnehmung seiner Verantwortung ist der *Container* ebenso wie das *Element* und der *Keeper* an *NeMO's Simulation Database* (NeSD) angeschlossen.

Für die Verwaltung von *Link-Keepern* existiert ein entsprechendes Gegenstück, der *Link-Container*. Analog zu einem *Link-Keeper* besitzt dieser keine Anbindung an *Factory*-Objekte, sondern interagiert mit dem *Trading Service* (s.o.), mit dem er standardmäßig verbunden ist.

6.4.2 Gesamtstruktur

Die Objekte des Modellmanagements werden unterteilt nach dem Kontext, in dem sie von Bedeutung bzw. in dem sie gültig sind. Demnach ergeben sich für *NeMO* die drei folgenden Teilbereiche:

1. *Local Nemo*

   Local Nemo umfaßt alle Objekte, die lokal auf einem Rechner von Bedeutung sind.

2. *Central Nemo*

   Eine übergeordnete Betrachtung und die zentrale Koordination erfolgt durch die Objekte von *Central Nemo*.

3. *Shared Nemo*

   Shared Nemo vereinigt alle Objekte, die einer gemeinsamen Nutzung von *Local* und *Central Nemo* unterliegen.

Local Nemo

Der durch *Local Nemo* beschriebene Objektbaum (vergleiche Abbildung 6.18) existiert für jeden Rechner, der in die Verwaltung durch das Modellmanagement einbezogen ist. Die Darstellung der Objekte und ihrer Beziehungen entspricht der UML-Notation (siehe [Mul97]).

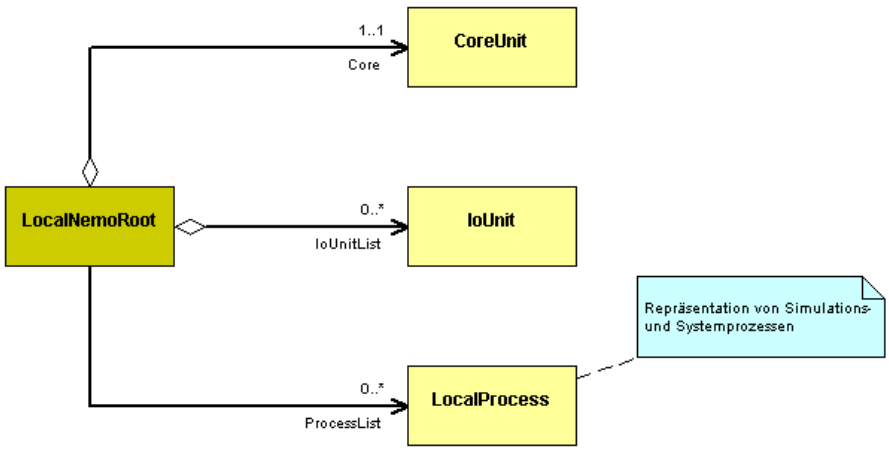


Abb. 6.18: Objektbaum *Local Nemo*

Die Bedeutung der Objekte ist teilweise schon angesprochen worden, sie soll aber des besseren Verständnisses wegen in dieser Gesamtbetrachtung noch einmal kurz wiederholt werden.

- *Local Nemo Root*

  Das Root-Element ist die Koordinations- und Anlaufstelle für die Repräsentation eines Rechners und enthält beispielsweise Informationen über das Betriebssystem oder welche Simulationsvorhaben („Sessions“) gerade am laufen sind. Das Root-Element muß für jeden Rechner vorhanden und betriebsbereit sein, anderenfalls kann er nicht vom Modellmanagement berücksichtigt werden.

- *Core Unit*

  Die *Core Unit* repräsentiert den Kern des Rechners aus der Sicht der Ressourcen (vergleiche Kapitel 6.1.1). Ein Rechner kann immer nur eine *Core Unit* besitzen.

- *Io Unit*

  Zur Ausstattung eines Rechners gehören ebenso Ein- und Ausgabeeinheiten (siehe Abschnitt 6.1.1). Diese werden durch Objekte mit dem Namen *„Io Unit“* beschrieben, wobei jedoch nur die Einheiten berücksichtigt werden, die für eine Simulation benötigt werden.

- *Local Process*

  Der *Local Process* ist der Teil des Abbildes eines Simulationsprozesses (Kapitel 6.3.2) hinsichtlich der Aspekte, die lokal auf dem ausführenden Rechner von Bedeutung sind. Durch *„Local Process“* Objekte werden auch die Systemprozesse repräsentiert, die in Abschnitt 3.3 vorgestellt worden sind und die keiner weiteren inhaltlichen Interpretation durch das Modellmanagement unterliegen.

Central Nemo

Mit der Bezeichnung *Central Nemo* sollen die Objekte gemeint sein, die die Zentrale des Modellmanagements ausmachen. *Central Nemo* ist deswegen auch der Teil des Modellmanagements, mit dem der Benutzer direkt in Berührung kommt. Für jedes Simulationsvorhaben („Session“) gibt es eine eigene Zentrale, die aber alle aus den gleichen Typen von Objekten aufgebaut sind (siehe Abbildung 6.19).

Wiederum sollen kurz die verschiedenen Objekte erläutert werden.

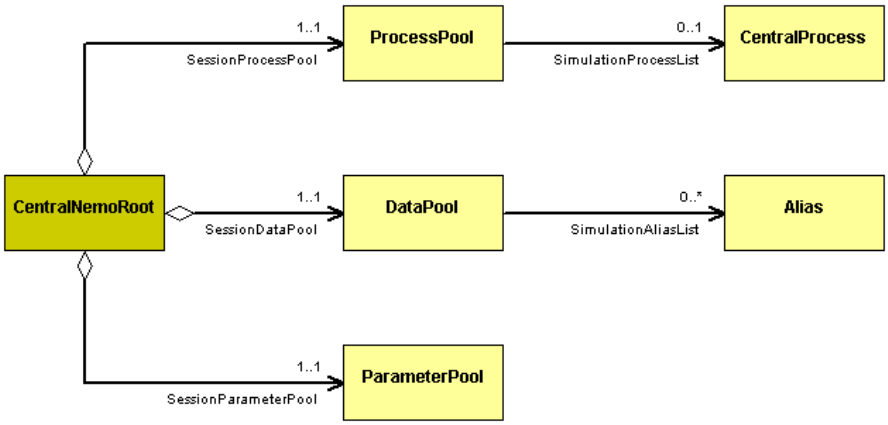


Abb. 6.19: Objektbaum *Central Nemo*

- *Central Nemo Root*

  Das zentrale Root-Element ist der Startpunkt eines jeden Simulationsvorhabens. Es verwaltet globale Einstellungen wie z.B. die Festlegung der Simulationswelt (siehe [Eng01]) und baut auch die Verbindung zum Worldmanager des Datenmanagements auf (vergleiche Kapitel 6.2), wodurch Prozesse und Daten zueinander zugeordnet werden können.

- *Process Pool*

  Der *Process Pool* verwaltet die Abbilder aller Simulationsprozesse und ermöglicht somit einen zentralen Überblick über die Prozeßstruktur.

- *Central Process*

  Die Repräsentation eines Simulationsprozesses aus der zentralen, übergeordneten Sicht heraus erfolgt mit Hilfe des *Central Process* Objekts. Es kennt beispielsweise die Signatur eines Prozesses, dessen Parameterbelegung, die Subprozesse, etc.. Für den Übergang auf die lokale Betrachtungsebene besitzt der *Central Process* eine nicht dargestellte Verbindung zum lokalen Prozeßabbild.

- *Data Pool*

  Der *Data Pool* ist eine Repräsentation des Datenmanagements aus *NeMO*'s Sicht. Es verwaltet, wie auch im Datenmanagement selbst, hauptsächlich die Aliase in der Simulation (vergleiche auch Kapitel 4.4).

- *Alias*

  Ein *Alias* Objekt, nach der Definition des Modellmanagements (siehe Abschnitt 6.2.2), besteht aus einem Namen, einer Angabe über den Variablentyp, den es repräsentiert, sowie der Summe aller Lese- und Schreibanmeldungen von Simulationsprozessen. Demzufolge sind die *Central Process* und die *Alias* Objekte miteinander verknüpft, was der Übersichtlichkeit wegen in Abbildung 6.19 nicht dargestellt worden ist.

- *Parameter Pool*

  Im *Parameter Pool* werden die Parameterbelegungen aller Prozesse zentral verwaltet, um den Benutzer bei der Zuweisung von Werten für Parameter neuer Prozesse unterstützen zu können, beispielsweise in der Frage, welche Parameterwerte nicht mehr benutzt werden dürfen.

Shared Nemo

Unter der Bezeichnung *Shared Nemo* werden alle Objekte verstanden, die von der Menge der *Local* und *Central Nemo's* gemeinsam genutzt werden. All diese Objekte kommen jeweils nur ein einziges Mal im gesamten Modellmanagement vor und sind in Abbildung 6.20 in der Übersicht dargestellt.

Die Bedeutung der einzelnen Objekte wird im folgenden kurz vorgestellt.

- *Shared Nemo Root*

  Über das Root-Element können die gemeinsam genutzten Objekte von den anderen über einen sogenannten *Naming Service* gefunden werden.

  Außerdem verwaltet es allgemeine Informationen für das *Central Nemo* Objekt, die dem Modellmanagement bekannten Typen von Betriebssystemen und Dynamic Link Libraries sowie die Festlegung der Referenzbausteine (vergleiche Aussagen über die Leistungsfähigkeit von Bausteinen; Kapitel 6.1).

- *Program Info*

  Program Info ist die interne Bezeichnung des Abbildes eines Simulationsprogramms, wie es in Abschnitt 6.3.2 vorgestellt worden ist.

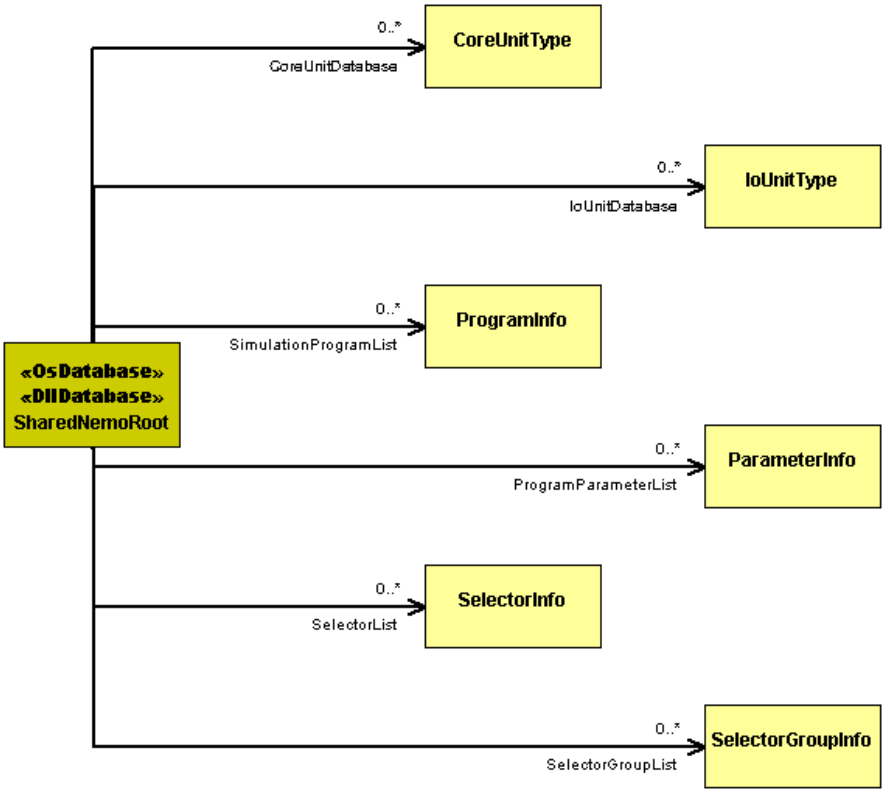


Abb. 6.20: Objektbaum *Shared Nemo*

- *Parameter Info*

  Unabhängig von einem Programm werden Parameter durch *Parameter Info* Objekte beschrieben. Diese Typinformation enthält den Namen, eine textliche Beschreibung, mögliche vordefinierte Wertebelegungen, Voreinstellung und einen Gültigkeitsbereich.

  Für die Festlegung der Parameter für ein Simulationsprogramm kann auf diese Definition zurückgegriffen werden, was den Eincheckvorgang (siehe Kapitel 6.3.4) eines Programms erleichtert, da viele Programme ähnliche Parameter verwenden.

- *Core Unit Type*

  Hierbei handelt es sich um vordefinierte Beschreibungen von Rechnerkernen (vergleiche Abschnitt 6.1), mit deren Hilfe die Repräsentation der vom Modellmanagement verwalteten Rechner erfolgt bzw. mit deren Hilfe, Anforderungen seitens der Programme an diese Rechner formuliert werden können.

Über die Typinformation wird die Leistungsfähigkeit des Bausteins in Zahlenform ausgedrückt und es wird festgehalten, ob es sich um die Referenzeinheit handelt.

- *Io Unit Type*

  Ein *Io Unit Type* Objekt ist im Hinblick auf Ein- und Ausgabebausteine das Analogon zum *Core Unit Type* Objekt.

- *Selector Info*

  Diese Art von Objekten wird von der Zentrale des Modellmanagements (*Central Nemo*) benötigt. Mit ihrer Hilfe werden in der Konfigurationsphase einer Simulation Programme bzw. Prozesse durch den Benutzer selektiert. Für eine nähere Erläuterung sei auf Kapitel 7.1.2 verwiesen.

- *Selector Group Info*

  Die oben beschriebenen *Selector Info* Objekte können in Gruppen angeordnet werden, wodurch die Gestalt und Funktionalität der Bedienoberfläche beeinflußt wird. Auch hier wird auf Abschnitt 7.1.2 verwiesen.

Die soeben dokumentierten Teilbäume des Modellmanagements werden unabhängig voneinander umgesetzt und entsprechend ihrer Bedeutung und Gültigkeit im Rechnernetzwerk verteilt (siehe Abbildung 6.21), so daß sie quasi „vor Ort“ sind, um ihrer Verantwortung gerecht werden zu können. Damit handelt es sich, genauso wie die von *NeMO* zu koordinierende Simulation, um eine verteilte Anwendung, in diesem Fall bestehend aus drei Komponenten.

Auf der anderen Seite ergibt sich daraus die Notwendigkeit, daß Objekte über Prozeß- und Rechnergrenzen hinweg miteinander kooperieren müssen, allein schon deswegen, um dem Menschen einen zentralen Zugriff auf das gesamte Modellmanagement zu ermöglichen.

Zu diesem Zwecke soll die Kommunikationsstruktur CORBA (vergleiche [Sie96] und [Sch]) verwendet werden, die als Basis des CORBA Component Model (siehe Kapitel 5.2.2) eingesetzt wird. Im Gegensatz zur Umsetzung des Komponentenmodells existieren Implementationen der *Common Object Request Broker Architecture* von verschiedenen Anbietern, die auch schon einer längeren Zeit der Erprobung unterzogen worden sind.

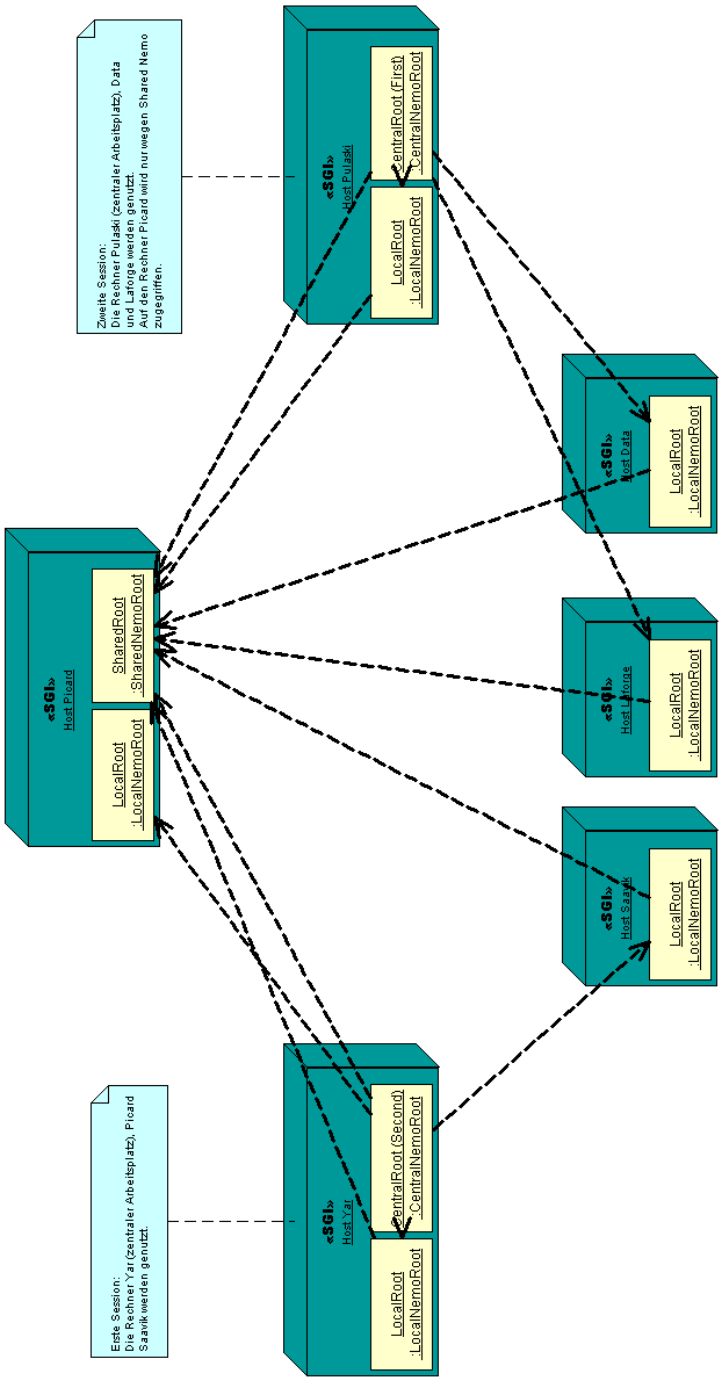


Abb. 6.21: Die Verteilung des Modellmanagements *NeMO*

In der daraus folgenden Konsequenz sind alle Basis-Objekte des Modellmanagements auch gleichzeitig CORBA-Objekte, so daß ihre Dienste über die Schnittstellen eines CORBA-Objekts von allen Rechnern aus, auf denen ein sogenannter *Objekt Request Broker* existiert, abgefragt werden können.

Außerdem bietet die CORBA-Infrastruktur zwei grundlegende Funktionalitäten, die von *Nemo's Model Organizer* zur Erfüllung seiner Aufgaben benötigt werden: ein *Naming* und ein *Trading Service*. Mit ersterem können CORBA-Objekte anhand eines Namens in einem Netzwerk aufgefunden werden (Stichwort *Shared Nemo Root*; s.o.) und mit dem *Trading Service* werden Objekte anhand ihres Typs für jeden Interessierten bekannt gemacht (Stichwort Links; s.o.).

Die ebenfalls mögliche Nutzung von DCOM (*Distributed Component Object Model* von *Microsoft*) bzw. Java/RMI (*Java/Remote Method Invocation* von *Sun Microsystems*) wurde nicht in Betracht gezogen, weil diese entweder nicht für alle Plattformen kostenfrei verfügbar sind oder nicht alle Programmiersprachen (in diesem Falle *C++*, mit deren Hilfe die Anbindung an das Datenmanagement erfolgt) unterstützen. Außerdem werden freie CORBA-Implementationen in der Regel als Quellcode zur Verfügung gestellt, so daß der Endnutzer die Möglichkeit einer Adaption oder einer Erweiterung besitzt.

6.5 Implementationsdetails

6.5.1 Allgemein

Bei der Umsetzung von *Nemo's Model Organizer* sind konsequent objekt-orientierte Ansätze (siehe auch [Sie96] und [Lou98]) angewandt worden. Für jedes der oben beschriebenen Objekte existiert eine Klassendefinition in der Programmiersprache *C++*, die die Eigenschaften der Objekte und die Methoden, wie darauf Einfluß genommen werden kann, bestimmt. Grundlage der Klassendefinitionen sind die im vorigen Abschnitt vorgestellten Basis-Objekte, die über Vererbung und Polymorphie an die spezifischen Aufgaben angepaßt worden sind.

Oberstes Ziel ist die Portabilität des Modellmanagements sowie eine möglichst effiziente und programmiererfreundliche Entwicklung. Aus diesem Grund sind in der Entwicklung verschiedene CASE-Tools (Computer Aided Software Engineering) begleitend zum Einsatz gekommen und es sind für spezielle Anwendungsbereiche existierende, plattform-übergreifende Programmbibliotheken verwendet worden.

Im einzelnen sind dies:

CASE-Tools

- Objekttechnologie Werkbank (OTW)

  OTW (siehe [OTW]) ist ein Modellierungstool, welches Objekte, deren Eigenschaften und deren Beziehungen mit Hilfe von Diagrammen nach der UML-Notation ([Mul97]) repräsentiert und damit auch gleichzeitig dokumentiert. Alle Klassen- oder IDL-Definitionen der Objekte in *NeMO*'s Rahmenwerk sind unter Einsatz dieses Hilfsmittels erarbeitet worden. Da Quellcode in reiner Textform (ASCII) vorliegt, konnte die *Objekttechnologie Werkbank* verwendet werden, obwohl sie nur für die Betriebsplattform *Microsoft Windows* zur Verfügung steht.

- TAO IDL Compiler

  Alle Objekte in *Nemo's Model Organizer* sind auch gleichzeitig CORBA-Objekte, welche mit Hilfe der Hochsprache IDL (*Interface Definition Language*) spezifiziert werden. Für die Hochsprache IDL existieren sogenannte *Language Mappings*, welche besagen, wie das Äquivalent einer IDL-Definition in der gewählten Programmiersprache, die für die Implementation eines CORBA-Objekts zum Einsatz kommen soll, aussieht.

  Im Falle von *NeMO* wird die Programmiersprache *C++* verwendet und die „Übersetzung" der IDL-Spezifikation übernimmt der *TAO IDL Compiler* (siehe [TAO]). Dieser erzeugt für jedes CORBA-Objekt eine *C++* Basisklasse, durch deren Ableitung dann die eigentliche Nutzer-Implementation erfolgt.

- Qt-Designer

  Der *Qt-Designer* ([QtD01]) ist ein graphisch-interaktives Design-Tool zur Auslegung und Gestaltung von Bedienoberflächen mit Hilfe der Programmbibliothek *Qt* (siehe unten). *NeMO*'s gesamte Benutzerschnittstelle (vergleiche Kapitel 7.1) ist mit diesem Hilfsmittel gestaltet worden, welches ähnlich wie der *TAO IDL Compiler* Klassendefinitionen produziert, die dann durch Ableitung für den Einsatz im Modellmanagement spezifiziert werden.

Programmbibliotheken

- Standard Template Library (STL)

  Die *Standard Template Library* ist ein durch ANSI-Standard spezifizierter Bestandteil der sogenannten *C++* Laufzeitbibliothek. Diese enthält grundlegen-

de, nicht in der eigentlichen Programmiersprache enthaltene Funktionalitäten, für die der Standard eine feste Klassendefinition vorsieht. Wie diese von einem Hersteller einer Programmierumgebung umgesetzt wird, ist, den Prinzipien der Objekt-Orientierung folgend, unbedeutend.

Als Teil der Laufzeitbibliothek stellt die *Standard Template Library* unter anderem generische Container- und Iteratoren-Objekte sowie Standard-Algorithmen zur Verfügung, die auf allen dem ANSI-Standard entsprechenden Plattformen eingesetzt werden können. Weiterhin wird durch die STL die objektorientierte Anbindung von Dateien in Form von sogenannten *I/O Streams* ermöglicht.

- The Adaptive Communication Environment (ACE)

  ACE ist ein sogenanntes objekt-orientiertes Netzwerk-Programmier „Toolkit" in *C++* (siehe [ACE]), welches Klassen und Methoden für die Kommunikation von Objekten auf einem Computer oder über das Netzwerk bereitstellt. Dabei werden alle betriebssystem-spezifischen Details durch das „Toolkit" abgefangen, so daß Quellcode auf der Basis von ACE auf allen Plattformen lauffähig ist, die durch das *Adaptive Communication Environment* unterstützt werden.

  Vorteil neben der Unterstützung aller gängigen Betriebssysteme ist die kostenfreie Nutzung und Verfügbarkeit des Quellcodes von ACE.

- The ACE ORB (TAO)

  TAO (siehe [TAO]) ist eine *open-source* Implementation der CORBA-Spezifikation ([COR]), welche ein Rahmenwerk zur Kooperationen von Objekten in einer verteilten Anwendung definiert. TAO nutzt auf transparente Art und Weise die Funktionalitäten von ACE (s.o), da es vom gleichen Anbieter stammt.

  Unabhängig von der Nutzung von ACE ist die Spezifikation der *Common Object Request Broker Architecture* an keine feste Plattform gebunden. Neben der reinen Kommunikationsstruktur stellt TAO auch allgemeingültige Dienste zur Verfügung, die ebenfalls Teil der CORBA-Spezifikation sind. *Nemo's Model Organizer* nutzt die bereits schon erwähnten *Naming* und *Trading Service*.

- Qt

  Qt ist eine plattformübergreifende Programmbibliothek zur Erstellung graphischer Benutzeroberflächen (GUI - Graphical User Interface). Zu diesem Zweck

kann ein Programmierer auf *C++* Klassendefinitionen für grundlegende Elemente einer Bedienoberfläche (Fenster, Buttons, Menüs, ...) zurückgreifen und diese zu einem individuellen GUI zusammenstellen, wie geschehen bei der Implementation von *NeMO*'s Bedienoberfläche (vergleiche Kapitel 7.1).

Qt ist für die meisten *UNIX*-Varianten frei und für *Microsoft Windows* kommerziell erhältlich. Es gibt jedoch besondere Konditionen für Universitäten, so daß für das Fachgebiet Flugmechanik und Regelungstechnik eine freie sogenannte „Evaluation Version" benutzt werden kann.

Einzige Ausnahme hinsichtlich einer Unabhängigkeit von einer bestimmten Plattform bildet der *Alias Manager* im Zusammenhang mit den Datenbankanbindungen des Modellmanagements (für eine detaillierte Beschreibung siehe Kapitel 7.2.3). Dieser wurde parallel zu *NeMO* entwickelt und wurde schon vor Fertigstellung von *Nemo's Model Organizer* benötigt, so daß die Datenbank-Software *Microsoft Access* zum Einsatz kam. Der *Alias Manager* ist jedoch mit einem Exportfilter versehen worden, der die Inhalte als reine Textdatei zur Verfügung stellt. Auf diese Weise kann der Inhalt auf andere Plattformen übertragen bzw. an nicht *Microsoft Office* Programme transferiert werden.

6.5.2 Spezifisch

Die Teilbereiche des Modellmanagements *Local* und *Shared Nemo* (s.o.) werden der Einfachheit halber in einem Prozeß zusammengefaßt, obwohl sie unabhängig voneinander existieren könnten. Da *Shared Nemo* nur einmal vorkommen darf, überprüft *NeMO* das nur ein einziger *Local Nemo* mit dem gemeinsamen Anteil kombiniert wird. Welcher der lokalen Vertreter des Modellmanagements dies ist, kann frei festgelegt werden. Es ist ebenfalls möglich, *Shared Nemo* online von einem *Local Nemo* zu einem anderen zu transferieren.

Jeder *Local Nemo* muß auf dem entsprechenden Rechner als Dauerdienst eingerichtet werden, der sich auf Anfrage zur Startzeit einer Simulation zu erkennen gibt, wodurch der Rechner vom Modellmanagement für die Simulation eingesetzt werden kann. Die Anfrage wird initiiert vom zweiten eigenständigen Prozeß des Modellmanagements, der die *Central Nemo* Objekte sowie die Bedienoberfläche für den Menschen in sich vereinigt. Der zentrale Arbeitsplatz wird für jede Simulations-Session gestartet, d.h. falls mehrere Simulationsvorhaben parallel laufen, existieren

auch mehrere Zentralen. Alle *Local Nemo's* und *Shared Nemo* werden dann gemeinsam genutzt.

Die schon erwähnte Bedienoberfläche ist vollkommen entkoppelt von den drei Bereichen des Modellmanagement aufgebaut. Genauso wie *Local*, *Shared* und *Central Nemo* nur über definierte Schnittstellen über Rechner- und Prozeßgrenzen miteinander interagieren, so gilt dies auch für das graphische User-Interface. Das Aussehen, die Gestaltung sowie die Bedienung der Bedienoberfläche werden im einzelnen im folgenden Kapitel bei der Vorstellung der Schnittstellen des Modellmanagements beschrieben.

Aufgrund der sauberen Trennung aller Verantwortlichkeiten und der Unabhängigkeit des Modellmanagements von alle Spezifika des Simulationsmodells ist *Nemo's Model Organizer* für eine eventuell spätere Erweiterung durchaus gerüstet. Prädestiniert für eine Adaption sind in diesem Zusammenhang

- die Rechnerrepräsentation,
- die Benutzeroberfläche und
- das Programm- bzw. das Prozeßabbild,

da dort mit der Zeit die größten Änderungs- und Erweiterungswünsche zu erwarten sein werden.

Die Entwicklungs- und Testplattform für das Modellmanagement ist das Betriebssystem *IRIX 6.5* von *Silicon Graphics*, da dies momentan noch die meistgenutzte Plattform in der Simulation am Fachgebiet Flugmechanik und Regelungstechnik darstellt. Andere Betriebssysteme wie *LINUX* und *Windows2000/NT* werden in Zukunft verstärkt zum Einsatz kommen, sind momentan aber noch relativ unbedeutend. Eine Portierung von *Nemo's Model Organizer* sollte aufgrund der obigen Ausführungen jedoch sichergestellt sein.

7 NeMO's Schnittstellen

In den Ausführungen über die Anforderungen an das Modellmanagement (vergleiche Kapitel 3) ist festgestellt worden, daß ein zentraler Zugang des Menschen zum Modellmanagement unerläßlich ist. Dieser Arbeitsplatz wird von *NeMO* in Form einer graphisch-interaktiven Benutzeroberfläche bereitgestellt.

Alle die Simulation betreffenden Interaktionen des Menschen erfolgen über diese Oberfläche, mit dessen Hilfe auch ein vollständiger Zugriff auf die Interna von *Nemo's Model Organizer* ermöglicht wird. Anhand der Gliederung der Benutzeroberfläche lassen sich wieder die drei großen Aufgabenbereiche Integration (*Checkin Wizard*), Konfiguration (*Model Setup Widget*) und Überwachung (*Monitor Widget*) erkennen.

Die im folgenden Abschnitt vorgestellte Umsetzung des *Graphical User Interface* (GUI) berücksichtigt ebenfalls die verschiedenen Nutzergruppen („normaler" Benutzer, Administrator und Entwickler) und deren Kenntnisse. In diesem Zusammenhang sei das Prinzip der Selektoren hervorgehoben. Die Selektoren assoziieren Simulationsaufgaben und/oder verwendete Hardware im Flugzeugcockpit mit den dazugehörigen Simulationsprozessen bzw. Prozeßstrukturen. Dadurch wird der „normale" Benutzer entlastet, da er in der Regel nur Kenntnisse über ersteres besitzt.

Unbedingt erforderlich als zweite Schnittstelle ist eine Datenbankanbindung zur Speicherung des Wissens über die Simulation. Wie diese implementiert worden ist, zeigt Kapitel 7.2. Hierbei läßt sich eine Dreiteilung von *NeMO's Simulation Database* erkennen. Die *Management Database* enthält alle internen Informationen über das Modellmanagement. Mit Hilfe der *Program Database* werden die in die Simulationsarchitektur integrierten Simulationsprogramme verwaltet und organisiert. Die *Communication Database* schließlich kann als ein interaktives *Interface Control Document* (ICD) angesehen werden, welches programm- bzw. prozeßübergreifend festhält, welche Daten in der Simulation ausgetauscht werden.

Die Datenbanken können plattformunabhängig existieren und werden vom Administrator, gegebenenfalls in Zusammenarbeit mit dem Entwickler, betreut. Dabei wird dieser durch die graphische Benutzeroberfläche des Modellmanagements unterstützt und angeleitet.

Beide Schnittstellen von *Nemo's Model Organizer* werden im folgenden eingehender erläutert.

7.1 Bedienoberfläche

Die Benutzeroberfläche kann grob nach den benötigten Kenntnissen zur Bedienung und damit nach den Nutzergruppen eingeteilt werden. Die Gruppe der einfachen Benutzer wird in der Regel nur mit dem *Startup Wizard* (Abschnitt 7.1.1) und dem *Model Setup Widget* (Abschnitt 7.1.2) in Berührung kommen.

Weitergehendes Wissen über den Aufbau des Modellmanagements und auch über die Simulationsmodule ist im Zusammenhang mit dem *Monitor Widget* (Kapitel 7.1.3) und dem *Checkin Wizard* (Kapitel 7.1.4) vonnöten, so daß diese in der Regel nur vom Administrator und gegebenenfalls vom Entwickler verwendet werden.

Eine detaillierte Beschreibung hinsichtlich Aussehen und Funktionalität der vier Teilbereiche ist Gegenstand der nachfolgenden Abschnitte.

7.1.1 *Startup Wizard*

Ein *Wizard* ist eine Bezeichnung für einen Dialog zur Abarbeitung sequentiell aufeinander folgender Schritte, wobei für jeden Schritt eine eigene Seite in der Bedienoberfläche existiert. In diesem Sinne ist *NeMO*'s *Startup Wizard* verantwortlich für ein geordnetes Hochfahren des Modellmanagements, indem er Schritt für Schritt den Menschen anleitet, welche Vorbereitungen getroffen werden müssen. Dies geschieht einerseits durch die vorhandenen Interaktionsmöglichkeiten als auch durch eine rein textliche Beschreibung dessen, was vom Nutzer erwartet wird. In Abbildung 7.1 ist zur Veranschaulichung die erste Seite des *Startup Wizard's* dargestellt.

Der Vorgang des Hochfahrens ist in vier Schritte unterteilt:

1. *Overview*

   Zu Beginn des Hochfahrens wird über den *Startup Wizard* im Netzwerk nach vorhandenen *Local Nemo's* und dem *Shared Nemo* (s.o.) gesucht, so daß der Benutzer erkennt, worauf er alles zurückgreifen kann.

   Es besteht auch die Möglichkeit, die Repräsentanten des Modellmanagements zu editieren bzw. den *Shared Nemo* von einem *Local Nemo* zu einem anderen wandern zu lassen. Dies ist jedoch bei entsprechender Voreinstellung durch den Administrator im Normalfall nicht nötig und eher für den versierten Anwender gedacht.

   Gleichzeitig wird dem Benutzer angezeigt, welche Simulationsvorhaben bereits parallel am Laufen sind und von welchem Rechner aus sie kontrolliert werden.

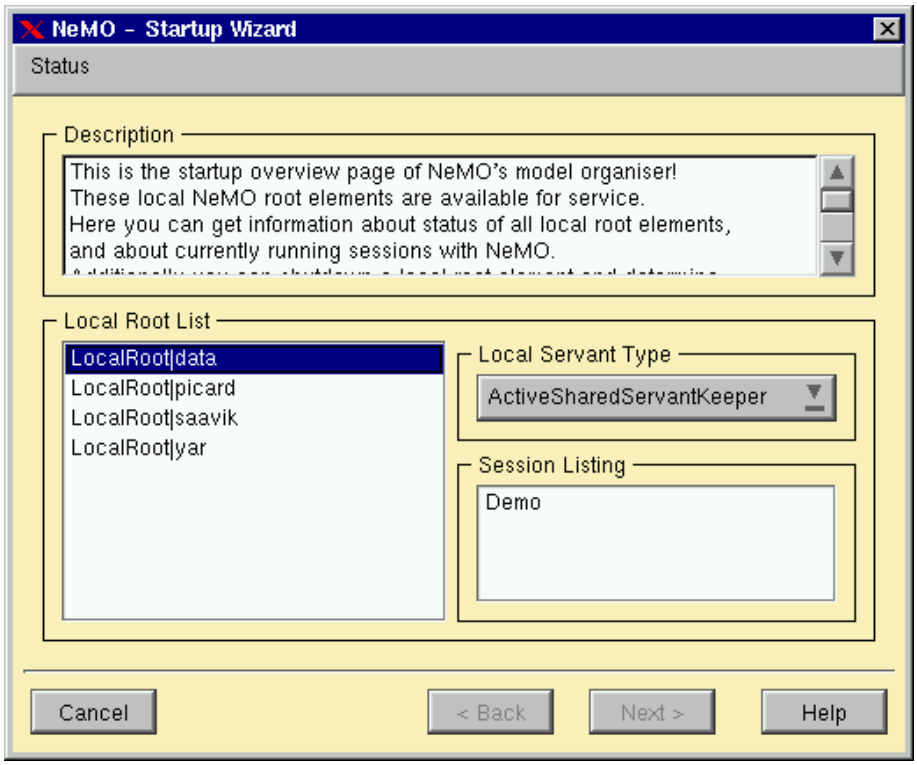


Abb. 7.1: *Startup Wizard* (First Page)

2. *Session Selection*

   Hierbei wird festgelegt, ob ein Simulationsvorhaben gestartet werden soll oder ob der Administrator das System einfach nur warten möchte. In beiden Fällen muß eine *Session Id* (beliebige, aber eindeutige Zeichenkette) angegeben werden, mit deren Hilfe die Identifikation im Netzwerk erfolgt.

   Die Vorbereitungen für eine Administrations-Session sind mit diesem Schritt abgeschlossen und der Administrator wird vom *Startup Wizard* zum *Monitor Widget* (siehe Abschnitt 7.1.3) weitergeleitet.

   Wird von einem Benutzer die Ausführung einer Simulation beabsichtigt, müssen die folgenden zwei Schritte noch ausgeführt werden.

3. *Host Selection*

   In dieser Phase wird vom Benutzer erfragt, welche der zur Verfügung stehenden Rechner für das Simulationsvorhaben zum Einsatz kommen sollen.

   Außerdem muß der Instruktor-Arbeitsplatz bestimmt werden, von wo aus der Benutzer die Simulationsprozesse und damit das Simulationsszenario kontrol-

lieren möchte. Im wesentlichen werden dort die Kontrollprozesse (vergleiche Kapitel 6.3.3) ausgeführt, welche die prozeßspezifische Koordination in der Simulation übernehmen. Der Instruktor kann vom zentralen Arbeitsplatz des Modellmanagements getrennt sein, muß aber nicht.

4. *Global Parameter*

   Als letzter Schritt der Startprozedur werden vom Benutzer einige Parameter abgefragt, die für die Simulation von globaler Bedeutung sind. Dazu gehört beispielsweise die Angabe der Simulationswelt ([Eng01]), mit deren Hilfe das Datenmanagement parallele Simulationsvorhaben voneinander trennt.

   Ein weiterer globaler Parameter ist die Festlegung der Datenbanken, die von den Anzeigeprozessen in der Simulation verwendet werden sollen. Je nach Forschungsprojekt können nämlich Datenbanken mit unterschiedlichen Inhalten in der Simulation zum Einsatz kommen.

7.1.2 *Model Setup Widget*

NeMO's *Model Setup Widget* (vergleiche Abbildung 7.2) ist die zentrale Schnittstelle für einen Benutzer in der Simulation, von der alle weiteren Aktionen ihren Ausgang nehmen.

Nachdem der Benutzer abhängig vom Simulationsvorhaben entschieden hat, welches Simulationsmodell zum Einsatz kommen soll, kann er mit der Hilfe des *Model Setup Widget's* die Umsetzung des Modells in Angriff nehmen. Dazu muß entschieden werden, welche Simulationsprozesse miteinander kombiniert werden sollen, um sie dann im Rechnernetzwerk verteilt auszuführen (Konfiguration der Simulation; vergleiche Kapitel 3.1.2).

Gleichzeitig bietet das *Model Setup Widget* eine überblicksartige Form der Überwachung der erzeugten Prozeßstruktur. Für eine gründlichere Einschätzung des Zustands der gesamten Simulation muß auf das *Monitor Widget* (Kapitel 7.1.3) übergegangen werden.

Obige Ausführungen machen deutlich, daß die Gruppe der Benutzer (siehe Abschnitt 3.2) das primäre Klientel für das *Model Setup Widget* darstellt, so daß versucht werden muß, deren Sichtweisen und Vorstellungen am ehesten gerecht zu werden.

Der Nutzer kennt in der Regel die Simulationsaufgabe, da er vorgibt, wie das Simulationsszenario auszusehen hat. Demzufolge weiß er, daß z.B. ein flugmechanisches Modell eines bestimmten Flugzeugtyps benötigt wird.

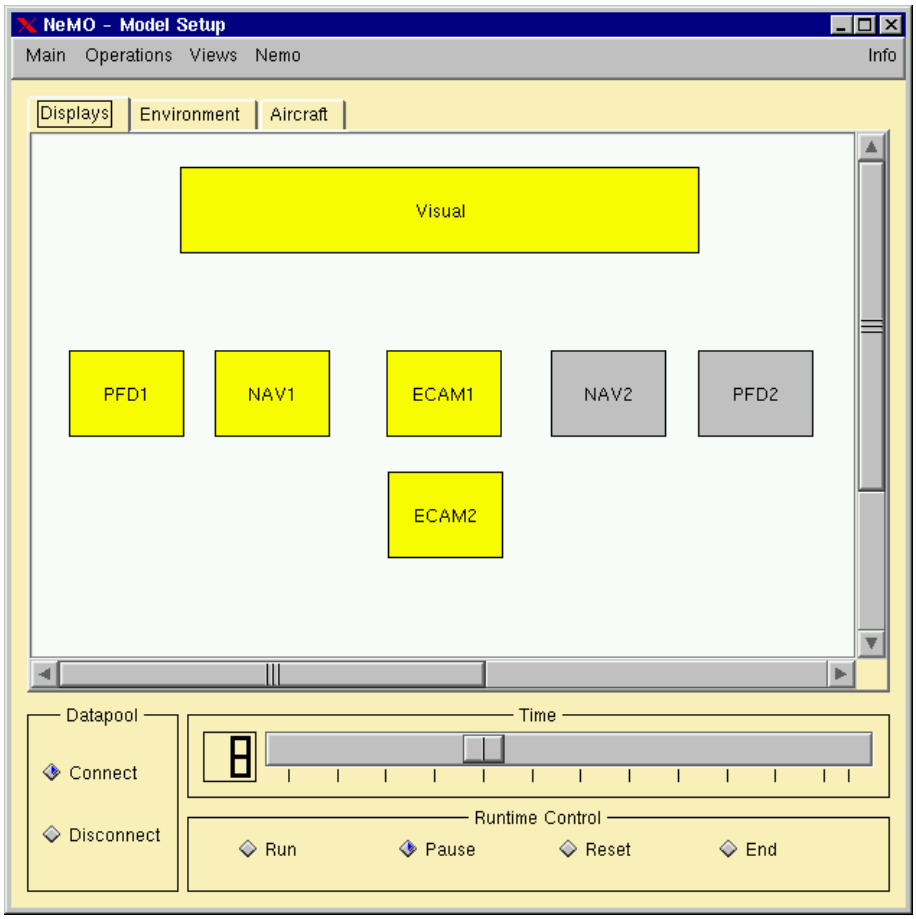


Abb. 7.2: *Model Setup Widget*

Desweiteren ist er meist vertraut mit der Hardware, mit der der Mensch in der Simulation interagiert. Dazu gehört unter anderem die Ausstattung eines Flugzeugcockpits, da die Forschungsprojekte am Fachgebiet Flugmechanik und Regelungstechnik sich mit der Verbesserung der Mensch-Maschine-Schnittstelle in Flugzeugcockpits beschäftigen. Die Assoziation eines spezifischen LC Display's beispielsweise mit der Anzeige von Flugzustandsinformationen stellt für den Benutzer deswegen kein Problem dar.

Welche Simulationsprozesse oder Prozeßstrukturen jedoch die dazugehörigen Aufgaben ausführen, darüber besitzt der Benutzer meist keine Kenntnisse. Das *Model Setup Widget* versucht daher eine Brücke zwischen der Vorstellungswelt des Benutzers und der verteilten Prozeßstruktur in der Simulation zu schlagen. Zu diesem Zwecke werden durch *Nemo's Model Organizer* sogenannte *Selektoren* und *Selektoren-Gruppen* eingeführt.

Programm Selektion

Ein *Selektor* ist ein graphisches Element, charakterisiert durch Name, Form und Farbgebung, welches mit einer überschaubaren Menge von Programmen oder Programmstrukturen assoziiert wird. Er ist allein zuständig für die Konfiguration, Ausführung und Überwachung der ihm zugeordneten Programme bzw. Prozesse, überschaut also nur einen relativ begrenzten Bereich der gesamten Simulation. *Selektoren* können in Gruppen angeordnet werden, so daß die Anordnung und Position der *Selektoren* zueinander als weiterer Informationsträger eingesetzt werden kann.

Über die graphischen Attribute, die der Mensch bei Bedarf interaktiv und dynamisch verändern kann, wird versucht, dem Benutzer die Bedeutung der mit einem Selektor assoziierten Programme zu vermitteln. Als Beispiel sollen die in einer Simulation zum Einsatz kommenden Anzeigen im Cockpit dienen (siehe Abbildung 7.3).

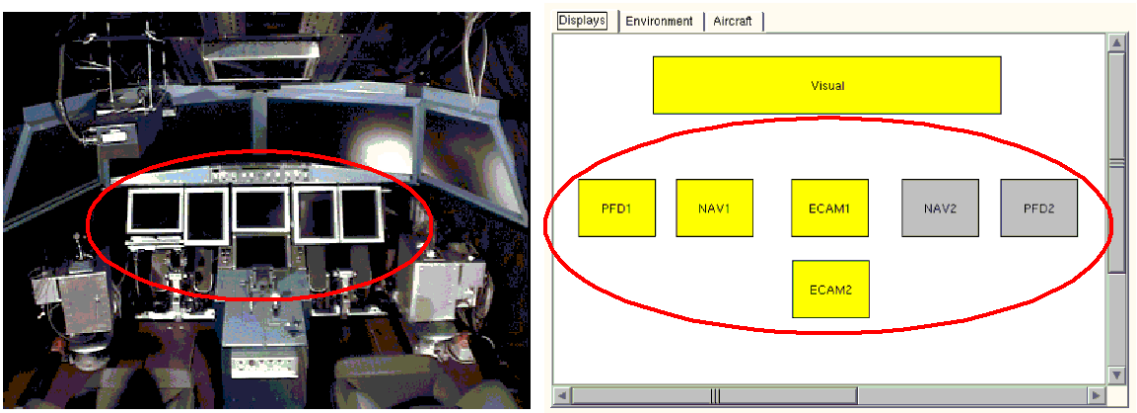


Abb. 7.3: Prinzip der Selektoren

Auf der linken Seite der Abbildung ist eine Photographie des in der Realität existierenden Forschungscockpits zu sehen. Deutlich zu erkennen sind die sechs großflächigen LC Display's in der Mitte des Flugzeugcockpits (angeordnet in Form des Buchstabens „T"), mit deren Hilfe die Piloten die wichtigsten Flugführungs- und Flugzustandsinformationen dargestellt bekommen.

Auf der rechten Seite ist zum Vergleich ein Ausschnitt des *Model Setup Widget's* abgebildet. Aufgrund der geometrischen Entsprechung kann der Benutzer unmittelbar schlußfolgern, daß dem Selektor links oben, mit dem Namen *PFD1*, diejenigen Simulationsprogramme zugeordnet sind, die für die Darstellung des *Primary Flight Display's* des *Captain* verantwortlich sind. Handelt es sich hierbei um mehr als ein

Simulationsprogramm, so können sie zu einer Programmgruppe zusammengefaßt werden, die der Benutzer auf einmal selektiert.

Die Hintergrundfarbe des Selektors dient dem *Model Setup Widget* als Indikator für eine rasche und übersichtliche Darstellung des Zustandes des Selektors. Im Normalzustand ist diese gelb, bei einem Fehlerfall wechselt sie dann zu rot und zeigt dadurch dem Benutzer an, daß dieser den Selektor eingehender untersuchen muß. Eine dritte Möglichkeit besteht darin, daß der Selektor inaktiv ist und nicht verwendet werden kann, weil er zum Beispiel keine Ressourcen zugeteilt bekommen hat. In diesem Fall ist der Hintergrund des Selektors grau eingefärbt.

Die Anordnung der Selektoren in Gruppen und deren Eigenschaften können in *NeMO's Simulation Database* (NeSD) abgelegt und zur Laufzeit wieder abgerufen werden. Über die Auswahl der Selektoren-Gruppen (*„Operations"*-Menü des *Model Setup Widget's*) kann ein Benutzer sich damit seine individuelle Schnittstelle zur Simulation konfigurieren. *Nemo's Model Organizer* kann daher für so unterschiedliche Vorhaben wie beispielsweise der Simulation mit dem in der Realität existierenden Forschungscockpit des Fachgebiets Flugmechanik und Regelungstechnik als auch der Simulation mit einem ebenfalls am Institut vorhandenen, virtuellen Cockpit (Virtual Reality) zum Einsatz kommen.

Welche Simulationsprogramme dem Selektor zugeordnet sind, kann der Benutzer mit Hilfe des *Program Selection Widget* betrachten (siehe Abbildung 7.4). Gleichzeitig werden auch hierüber die auszuführenden Prozesse selektiert.

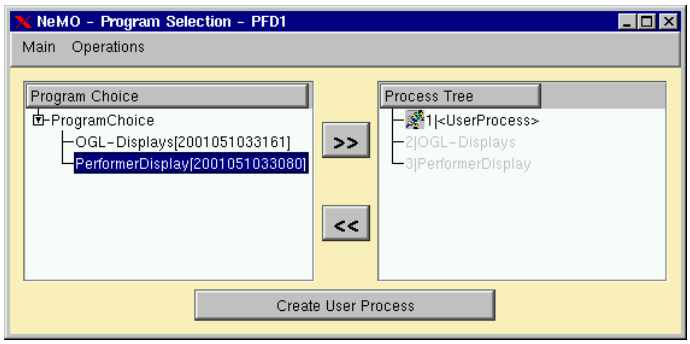


Abb. 7.4: *Program Selection Widget*

In der linken Hälfte des *Program Selection Widget's* befindet sich eine Baumdarstellung aller von diesem Selektor aus anwählbaren Simulationsprogramme. Für den Fall, daß der Nutzer eingehendere Informationen über das Programm benötigt als nur den angezeigten Namen, kann er über ein Kontext-Menü zu einer ausführlichen

Darstellung der Programmeigenschaften gelangen, wie sie in Kapitel 6.3.2 unter der Bezeichnung Programmabbild vorgestellt worden ist.

Der Programmbaum berücksichtigt die Existenz verschiedener Versionen eines Simulationsprogramms und ermöglicht dem Benutzer jede Version anhand ihrer Versionskennzeichnung für die Simulation auszuwählen. Die Versionskennzeichnung ist eine Zahl, die aus dem Datum und der Uhrzeit zusammengesetzt ist, zu der die Version des Simulationsprogramms in das Modellmanagement „eingecheckt" worden ist. Die Unterschiede und Besonderheiten der Versionen werden in den Programmeigenschaften dokumentiert.

Die rechte Hälfte des *Program Selection Widget* enthält eine Darstellung der bereits selektierten Prozesse, in die wiederum eine graphische Zustandsanzeige integriert worden ist. Prozeßnamen in roter Schrift deuten einen Fehler an, graue Schriftzüge dagegen sagen aus, daß der Prozeß (noch) nicht gestartet werden kann. Gründe hierfür könnten beispielsweise sein, daß der Prozeß erst auf einem Rechner plaziert werden muß oder das die Parameter des Prozesses noch nicht festgelegt worden sind. Wird der Prozeß in schwarzer Schrift angezeigt, so kann er gestartet werden oder befindet sich bereits in Ausführung. In letzterem Falle wird dem Prozeßnamen ein „Runner"-Icon (eine miniaturisierte Darstellung eines Läufers) vorangestellt.

Analog zur linken Seite können über ein Kontext-Menü die Eigenschaften eines Prozesses abgerufen werden, welche vom Inhalt im wesentlichen mit den beiden Formen der Abbilder eines Simulationsprozesses (lokal und central; siehe Kapitel 6.3.2) übereinstimmen.

Ausgewählte Prozesse und deren Parameterbelegungen können für jeden Selektor getrennt abgespeichert und wieder geladen werden, um den Benutzer die Möglichkeit zu geben, auf bereits früher verwendete Simulationsmodelle zurückzugreifen, anstatt alles immer von Grund auf neu zu konfigurieren.

Durch den Einsatz von Selektoren und Selektoren-Gruppen wird die Komplexität für den Benutzer so reduziert, daß dieser in die Lage versetzt wird, die Simulationsumgebung auch ohne größere Kenntnisse über die Prozeßstruktur, die zur Ausführung des Simulationsmodells benötigt wird, zu konfigurieren. Man kann sich das so vorstellen, als ob das tiefergehende Wissen quasi in den Selektoren gespeichert worden ist, beispielsweise vom Administrator und/oder einem versierten Experten-Benutzer.

Übersichten

Zusätzlich zur eher lokalen Form der Betrachtung der Simulation durch die Selektoren werden vom *Model Setup Widget* verschiedene Übersichtsdarstellungen angeboten. Die erste enthält eine vollständige Liste aller Simulationsprozesse, die im Rahmen des Simulationsvorhabens über die vorliegenden Bedieneroberfläche selektiert worden sind (Abbildung 7.5), während die zweite den schon in Abschnitt 4.4 und 6.4.2 angesprochenen Datenbaum der Simulation graphisch aufbereitet (Abbildung 7.6).

Beide sind über den Eintrag *„Views“* in der Menü-Leiste des *Model Setup Widget's* zu erreichen.

Abgerundet werden die Möglichkeiten der übergeordneten Betrachtung durch eine Ansicht der in der Simulation eingesetzten Ressourcen (*Ressource Selection View*), welche in der Abbildung 7.7 zu sehen ist. Über diese Oberfläche kann der Benutzer die Auswahl der benutzten Computer verändern und Rechte zur Benutzung vergeben. Letzteres bedeutet, daß jeder der oben beschriebenen Selektoren eindeutig mitgeteilt bekommt, welche Rechnerbausteine (vergleiche auch Abschnitt 6.1) oder Teile derer von ihm verwendet werden dürfen. Aus Gründen der Benutzerfreundlichkeit können diese Einstellungen gespeichert werden, so daß die Verteilung der Ressourcen nicht für jedes Simulationsvorhaben von neuem entschieden werden muß.

Mit Hilfe der oben beschriebenen Ansichten bietet das Modellmanagement für jeden der drei wesentlichen Bestandteile einer Simulation, **Prozesse**, **Daten** und **Ressourcen**, eine eigene, übergeordnete Betrachtungsebene.

Ablaufsteuerung Simulation

Nach den in Abschnitt 6.3.3 vorgestellten Richtlinien zur Ablaufsteuerung der Simulation ist das Modellmanagement auch gleichzeitig oberster Kontrollprozeß. Zu diesem Zwecke ist im unteren Teil des *Model Setup Widget's* eine rudimentäre Simulationssteuerung integriert.

Bestandteil der Kontrolle sind die Kommandos *Run*, *Pause*, *Reset* und *End* sowie die Vorgabe der globalen Simulationszeit (beispielsweise zur Einstellung, ob Tag- oder Nachtflug-Bedingungen). Da die Ablaufkontrolle über das Datenmanagement erfolgt, muß das *Model Setup Widget* mit dem Datenpool verbunden werden.

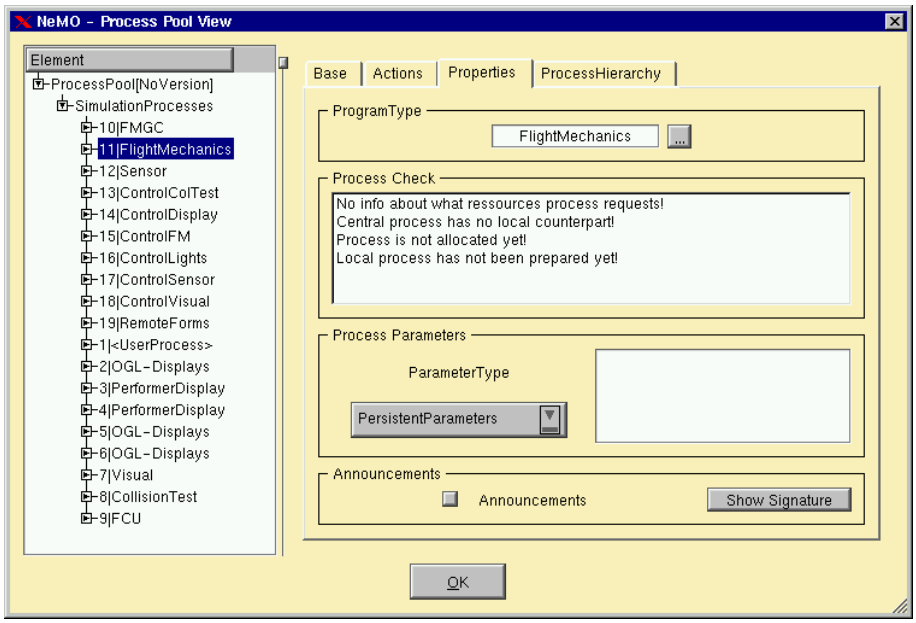


Abb. 7.5: *Process Pool View*

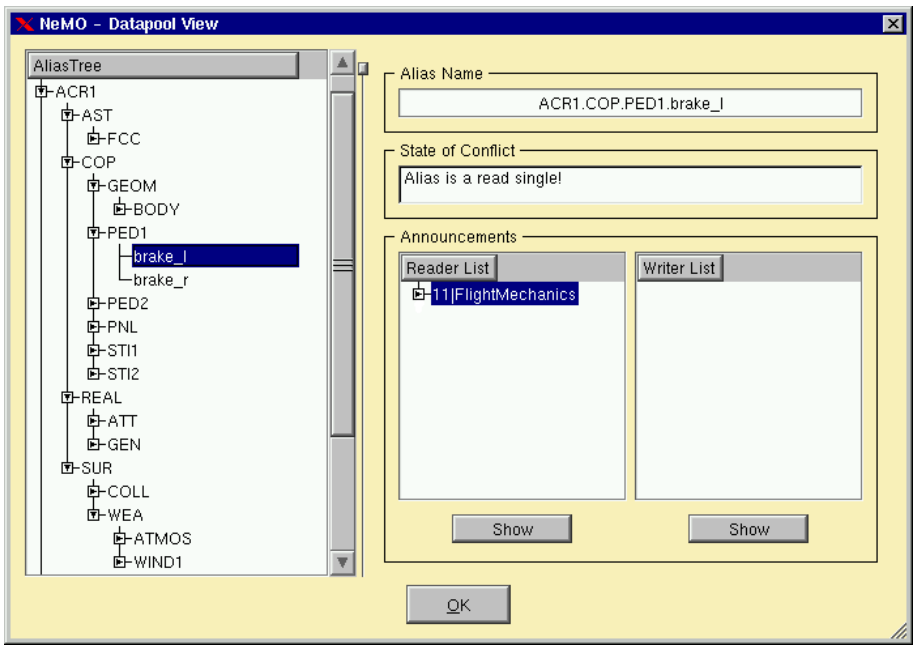


Abb. 7.6: *Data Pool View*

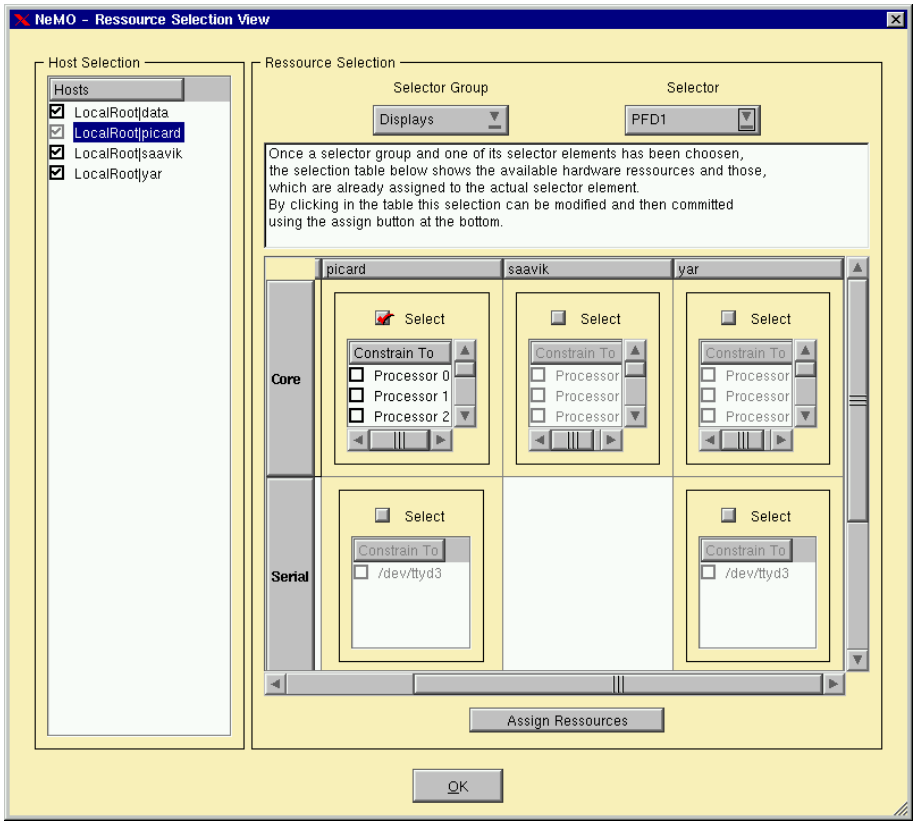


Abb. 7.7: *Ressource Selection View*

7.1.3 *Monitor Widget*

Das Ziel des Modellmanagements ist neben der Unterstützung des Benutzers in der Konfiguration und Ausführung einer Simulation die Bereitstellung eines zentralen Leitstands, von dem aus alle Beteiligten des Simulationsvorhabens überwacht werden können. Daraus werden die folgenden Forderungen abgeleitet:

- Jedes Objekt des Modellmanagements muß individuell zugänglich sein.
- Alle Eigenschaften der Objekte müssen vom Menschen begutachtet und gegebenenfalls verändert werden können.

Die Forderungen bedeuten eine Aufhebung der Transparenz des Modellmanagements, so daß die Interna der Gesamtstruktur zum Vorschein kommen. Der Ansatz zur Umsetzung dieser Forderungen ist zweigeteilt:

1. Erweiterung der Objektdefinition um eine graphische Repräsentation

   In Erweiterung zur Definition von Objekten durch Informationen und Operationen, die auf die Informationen angewendet werden können, wird für alle Objekte eine graphische Repräsentation gefordert. Das heißt, daß für jeden Objekttyp auch eine graphische Oberfläche existiert, mit deren Hilfe man mit dem Objekt interagieren kann. Die Grundstruktur der Benutzeroberfläche ist für alle Objekte gleich.

2. Standardisierte Form von Beziehungen zwischen Objekten

   Beziehungen zwischen Objekten werden über eine fest vorgegebene und unveränderliche Substruktur realisiert (vergleiche Kapitel 6.4.1). Dadurch wird eine standardisierte Form der Navigation von Objekt zu Objekt ermöglicht.

Im Endeffekt erhält man als Leitzentrale für das Modellmanagement das in Abbildung 7.8 dargestellte *Monitor Widget*.

Der linke Teilbereich des *Monitor Widget's* enthält eine der Gesamtstruktur von *Nemo's Model Organizer* entsprechende Baumdarstellung. Man erkennt die bereits in Kapitel 6.4.2 vorgestellten Teilbäume *Local*, *Central* und *Shared Nemo*.

Alle Objekte im Baum sind über *NeMO*'s Basis-Objekte *Element*, *Keeper* und *Container* miteinander verknüpft (vergleiche die Ausführungen in Abschnitt 6.4.1), was

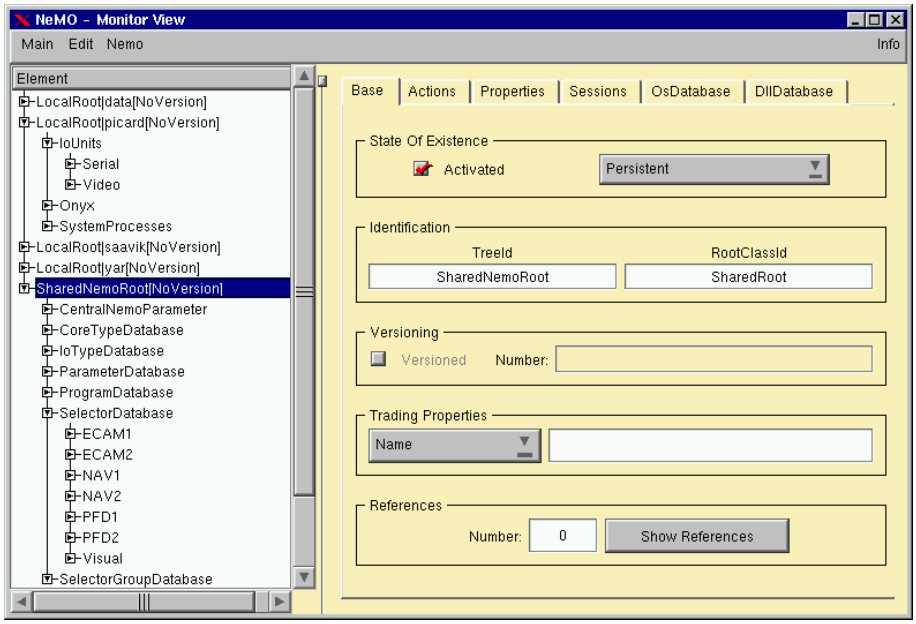


Abb. 7.8: *Monitor Widget*

vom *Monitor Widget* genutzt wird, um von einem Objekt zum anderen zu gelangen. Die gesamte Baumstruktur und damit das Modellmanagement kann an dieser Stelle den Bedürfnissen des Bedieners entsprechend verändert und adaptiert werden.

Die ebenfalls in Abschnitt 6.4.1 vorgestellten *Links* als besondere Form der Beziehung zwischen Objekten sind in der Baumdarstellung nicht enthalten, sondern werden auf der rechten Seite des *Monitor Widget's* gesondert angezeigt. Dort sind allgemein die Eigenschaften eines Objektes abgebildet, entsprechend der in Zusammenhang mit der Objektdefinition festgelegten graphischen Repräsentation des Objektes. Der Übersichtlichkeit halber werden die Objekteigenschaften in Kategorien unterteilt.

Für ein volles Ausschöpfen der Funktionalität des *Monitor Widget's* sind Kenntnisse über die Interna des Modellmanagements notwendig. Demzufolge ist dieses in erster Linie als Hilfsmittel für den Administrator, den Entwickler oder einem Experten-Nutzer gedacht, und weniger für den „normalen“ Benutzer. Der Ausrichtung auf die Zielgruppe folgend werden alle nicht unmittelbar im Zusammenhang mit einem Simulationsvorhaben benötigten Tätigkeiten über das *Monitor Widget* ausgeführt. Dazu zählen auch die Verwaltungsaufgaben, die *offline* - nicht während eines Simulationsvorhabens - anfallen, wie beispielsweise die Wartung der Datenbanken des Modellmanagements.

Mit Hilfe der dem *Monitor Widget* zugrundeliegenden Ideen kann der Mensch quasi Bestandteil des Modellmanagements werden und mit allen Objekten kooperieren, sie dadurch beeinflussen, Aktionen initiieren und die Erfüllung der Aufgaben durch das Modellmanagement überprüfen. Man kann sich den Menschen in diesem Zusammenhang als übergeordneten, intelligenten Beobachter von *Nemo's Model Organizer* vorstellen.

7.1.4 *Checkin Wizard*

Der *Checkin Wizard* (siehe Abbildung 7.9) ist ein graphisch-interaktiver Dialog zur Anleitung des Bedieners bei der Durchführung des in Kapitel 6.3.4 beschriebenen Eincheckvorgangs. Er sorgt dafür, daß die Teilschritte des Vorgangs in der richtigen Reihenfolge und unter Beachtung der Voraussetzungen abgearbeitet werden.

Erreichbar ist der *Checkin Wizard* über das oben beschriebene *Model Setup Widget* (Eintrag im Menü *Operations*), da es durchaus vorkommen kann, daß für die während des Eincheckens vorgenommenen Testläufe andere Simulationsprogramme als Umgebung benötigt werden.

In Abbildung 7.9 ist die zweite Seite des Dialogs angezeigt, über die die Eigenschaften des Programms abgefragt werden. Wie auch schon beim *Startup Wizard* (Kapitel 7.1.1) wird zusätzlich im oberen Teil in textlicher Form beschrieben, was vom Benutzer erwartet wird und unter welchen Bedingungen zur nächsten Seite übergegangen werden kann.

Die Eigenschaften eines Simulationsprogrammes können auch vollständig über das *Monitor Widget* (siehe Abschnitt 7.1.3) eingegeben werden, in diesem Fall aber ohne große Anleitung. Deswegen erfordert ein Einchecken „von Hand“ wesentlich mehr Verständnis über die Interna des Modellmanagements und ist nur für einen versierten Nutzer anzuraten.

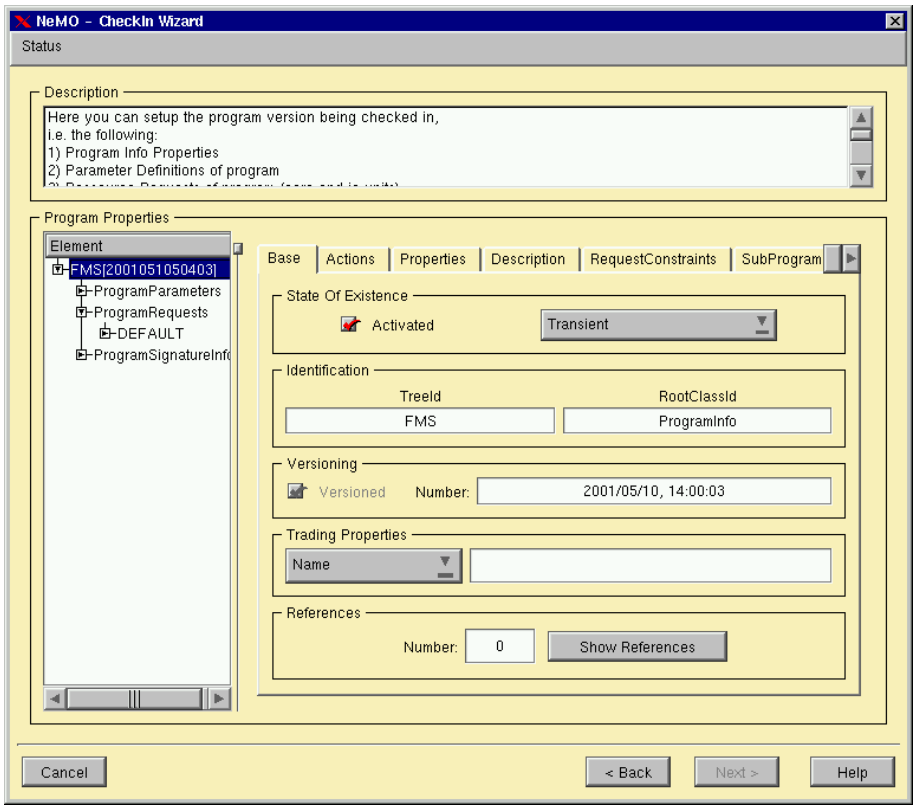


Abb. 7.9: *Checkin Wizard* (Second Page)

7.2 Datenbankanbindung

NeMO's Simulation Database (NeSD) ist das „Gedächtnis“ des Modellmanagements. Durch die Verknüpfung von *Nemo's Model Organizer* mit einer Datenbank wird erreicht,

- daß die Simulation am Fachgebiet Flugmechanik und Regelungstechnik von Menschen mit unterschiedlichem Wissensstand verwendet werden kann,
- daß die Bedienung einer Simulation im Rahmen der verteilten Architektur *DSPA* vereinfacht und weitgehend automatisiert wird und
- daß das Wissen über die Simulation nicht mit dem Ausscheiden von Mitarbeitern am Institut verloren geht, sondern auf definierte Art und Weise erhalten bleibt.

Die Datenbank des Modellmanagements wird aufgrund der verwalteten Informationen in drei eigenständige Bereiche aufgeteilt, so daß der Begriff *NeMO's Simulation*

Database eher als Sammelbegriff für drei Teil-Datenbanken zu verstehen ist:

1. *NeMO's Management Database (NeMD)*
2. *NeMO's Program Database (NePD)*
3. *NeMO's Communication Database (NeCD)*

Weswegen diese Aufteilung getroffen worden ist, wird anhand einer Beschreibung der Datenbanken in den folgenden Abschnitten erläutert werden.

7.2.1 *NeMO's Management Database*

NeMO's Management Database (NeMD) ist die persistente Darstellung der Objekte der beiden Teilbäume *Local* und *Shared Nemo* (vergleiche Beschreibung in Kapitel 6.4.2).

Der dritte Teilbaum des Modellmanagements, *Central Nemo*, existiert nur zur Laufzeit und wird nicht abgespeichert. Bei Bedarf kann jedoch ein Benutzer Logdateien des Simulationsvorhabens erzeugen, in denen festgehalten wird, welche Simulationsprozesse gestartet worden sind (*„Process Pool Listing“*) und welche Daten sie miteinander ausgetauscht haben (*„Data Pool Listing“*).

Der Inhalt der Management-Datenbank wird den Ausführungen folgend unterschieden nach Informationen, die lokal für einen Rechner gelten, und nach gemeinsam genutzten Informationen:

- Lokale Information

  Damit ist im wesentlichen die Rechnerrepräsentation gemeint, also Informationen über das Betriebssystem, über den Rechnerkern und über die Ein- und Ausgabebausteine. Zusätzlich können getrennt für jeden Rechner noch Systemprozesse (siehe Abschnitt 3.3) registriert werden. Welche dies sind und wie sie gestartet werden müssen, wird ebenfalls in der Datenbank festgehalten.

- Gemeinsam genutzte Information

  Darunter fallen im wesentlichen, wie schon vorher ausgeführt, die Beschreibungen von Parametern und Programmen, Typenlisten für die Repräsentation der Rechner, die Gestaltung der User-Interfaces über Selektoren und Selektoren-Gruppen sowie einige sonstige Informationen von allgemeiner Bedeutung.

Darin eingeschlossen sind beispielsweise vorkonfigurierte Simulationsmodelle und die Verteilung der Ressourcen auf die auszuführenden Prozesse.

NeMO's Management Database (NeMD) ist für die Erfüllung der Aufgaben des Modellmanagements unbedingt erforderlich und deswegen als einzige Teil-Datenbank über die in Kapitel 6.4.1 beschriebene Substruktur direkt in *Nemo's Model Organizer* integriert (Datenbankanbindung und -koordination über die Basis-Objekte *Element*, *Keeper* und *Container*).

Vom Format her handelt es sich bei der Datenbank um ein reines Textfile (vergleiche Abbildung 7.10), in dem die Baumhierarchie über geschachtelte Bereiche abgebildet wird. Die Bereiche werden durch „Keywords" eingeleitet und sind in geschweiften Klammern eingeschlossen. Am Zeilenanfang und -ende können Kommentare eingefügt werden, indiziert über die Zeichenfolge „`##`". Die Interpretation der „Keywords" und der Daten in den geschweiften Klammern erfolgt durch die betroffenen Objekte selbst.

Der Vorteil einer reinen ASCII-Datei ist der, daß sie auf jede Plattform portiert und daß zum Ein- und Auslesen auf standardisierte Mechanismen aus der *C++*-Laufzeitbibliothek zurückgegriffen werden kann (siehe dazu auch Abschnitt 6.5.1). Es wird keine extra Datenbanksoftware benötigt, die eventuell mit Beschränkungen hinsichtlich Verfügbarkeit und Portabilität versehen wäre.

Außerdem ermöglicht die reine Textdarstellung, daß der Zustand der Objekte vom Menschen, beispielsweise dem Administrator, *offline* analysiert werden kann. In Fehlerfällen kann nämlich häufig zur Laufzeit keine eingehendere Untersuchung durchgeführt werden, sondern muß im Nachhinein erfolgen. Allgemein wird durch die Lesbarkeit der Datenbank die Transparenz des Modellmanagements erhöht.

Da der gemeinsam genutzte Objektbaum *Shared Nemo* mit einem beliebigem *Local Nemo* assoziiert werden kann (siehe Kapitel 6.5.2), wird *NeMO's Management Database* (NeMD) in zwei, den Inhalten entsprechenden Dateien aufgeteilt, für die die Root-Elemente von *Local* und *Shared Nemo* die Verantwortung tragen. Sie kennen die Verzeichnisstruktur, in der die Dateien abgelegt werden, und über sie wird das Lesen und Schreiben der Datenbank initiiert.

7.2.2 *NeMO's Program Database*

In der Management-Datenbank (s.o.) werden unter anderem die Abbilder von Simulationsprogrammen gespeichert, aber nicht die Programme selbst.

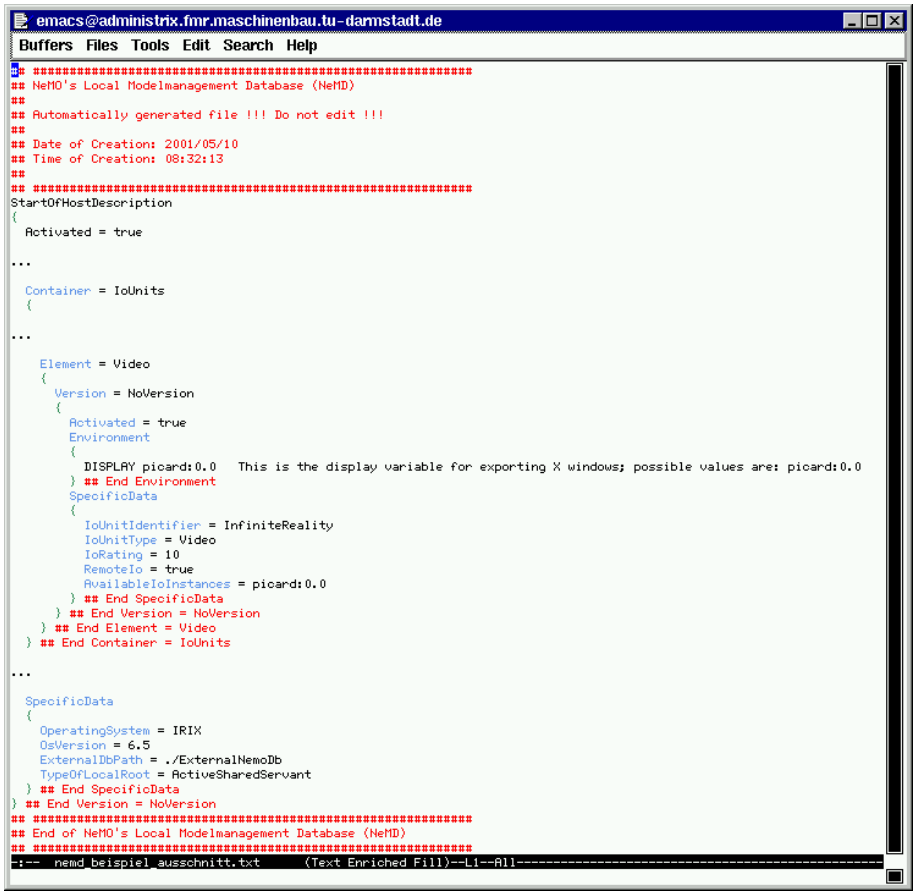


Abb. 7.10: *NeMO's Management Database* (NeMD) - Ausschnitt

Die Programme benötigen zur korrekten Ausführung ihrer Selbst eine individuell verschiedene Verzeichnisstruktur von Dateien, auf die die erzeugten Prozesse zur Laufzeit zugreifen. Der Aufwand, eine solche Struktur den Prozessen gegenüber zu emulieren, ist für die Ziele der vorliegenden Arbeit unverhältnismäßig hoch.

Deswegen werden die Simulationsprogramme nicht in einer integrierten Datenbank verwaltet, sondern über eine von *Nemo's Model Organizer* vorgegebene, externe Verzeichnisstruktur, die als *NeMO's Program Database* (NePD) bezeichnet wird. Abbildung 7.11 enthält eine Darstellung des Aufbaus der Programm-Datenbank, während in Abbildung 7.13 ein explizites Beispiel vorgestellt wird. Es sei darauf hingewiesen, daß *NeMO's Program Database* nur die Dateien zur Ausführung eines Programms enthält und keine Quellcode-Verwaltung darstellt.

Die Struktur von *NeMO's Program Database* wird relativ zu einem Referenzpunkt („Mount-Point“) definiert. Der „Mount-Point“, *NeMO*-intern als *External Root Path* bezeichnet, ist nichts anderes als eine Pfadangabe, die den Ort der Speicherung,

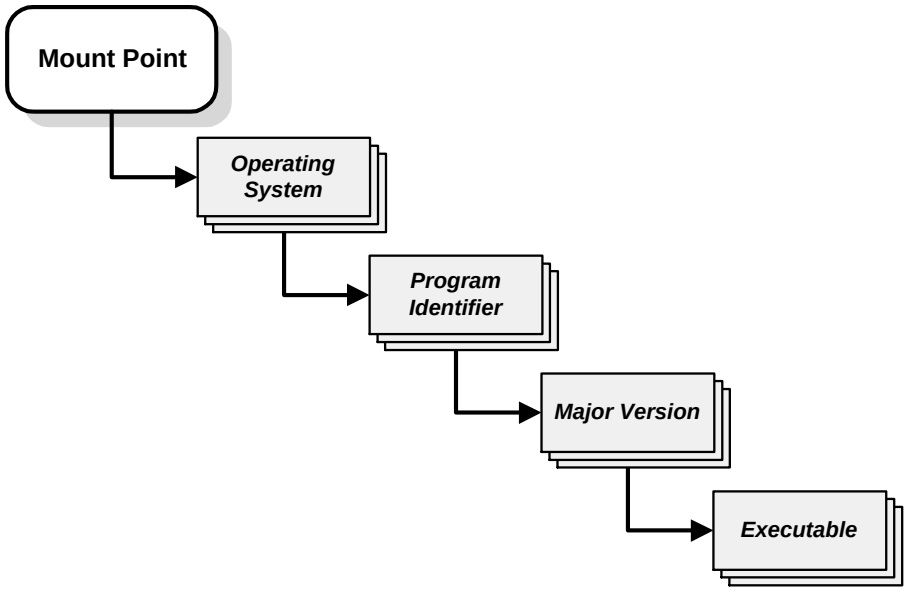


Abb. 7.11: *NeMO's Program Database* (NePD) - Strukturdiagramm

beispielsweise auf einer Festplatte, festlegt, und kann individuell für jeden Rechner, auf dem *Nemo's Model Organizer* installiert worden ist, angegeben werden.

Damit besitzt der Administrator der verteilten Simulationsarchitektur *DSPA* die Möglichkeit, die Programm-Datenbank zentralisiert oder dezentralisiert aufzubauen. Im ersten Falle würden alle Simulationsprogramme auf einer Festplatte gespeichert, auf die von allen Rechnern des Netzwerkes aus zugegriffen werden kann. Im letzteren Fall würde Teilbereiche der Datenbank auf den Rechnern lokal abgelegt werden, wo sie entsprechend dem Einsatz in der Simulation am häufigsten benötigt würden.

Die Verwaltung einer dezentralisierten Datenbank bedeutet jedoch für den Administrator erhöhten Aufwand, weil er die Konsistenz und Eindeutigkeit der in der Datenbank enthaltenen Simulationsprogramme sicherstellen muß, z.B. um zu verhindern, daß ein und dasselbe Programm nicht an zwei verschiedenen Stellen mit unterschiedlichen Informationen vorhanden ist.

Wie anhand des Strukturdiagramms in Abbildung 7.11 zu erkennen, werden unterhalb des Referenzpunktes vier verschiedene Ebenen definiert:

1. *Operating System*

   Die erste Ebene legt das Betriebssystem fest. Alle Simulationsprogramme, die unterhalb eines solchen Eintrags zu finden sind, können nur auf der entsprechenden Plattform ausgeführt werden.

2. *Program Identifier*

   Der *Program Identifier* ist der globale Bezeichner für ein Simulationsmodul in all seinen Versionen. Er sollte, wenn möglich, für den Menschen verständlich die Bedeutung des Moduls vermitteln und beispielsweise keine Abkürzung sein.

3. *Major Version*

   Diese Ebene beschreibt die Hauptversion, zu der eine andere Version eines Simulationsprogrammes kompatibel ist. Kompatibel heißt zumindest, daß die neue Version die gleiche Verzeichnisstruktur wie eine ältere erwartet. Die älteste Version, bis zu der diese Voraussetzung erfüllt ist, wird als *Major Version* bezeichnet.

   Zum besseren Verständnis sei an dieser Stelle kurz vorgestellt, wie sich eine Versionskennzeichnung zusammensetzt (siehe Abbildung 7.12): es handelt sich um eine Zahl, die aus dem Jahr, dem Monat, dem Tag und der Uhrzeit erzeugt wird, und zwar derart, daß eine kleinere Zahl einer älteren Version entspricht.

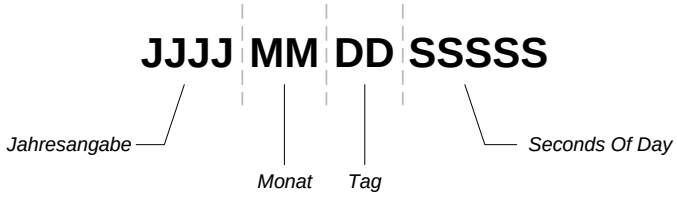


Abb. 7.12: Aufbau einer Versionskennzeichnung

   Es können jedoch auch andere, inhaltlich orientierte Kriterien bezüglich Kompatibilität definiert werden, so daß eine Programmversion durchaus die gleiche Verzeichnisstruktur wie vorherige Versionen voraussetzen kann, aber trotzdem diese nicht als Hauptversion angibt. Letztendlich wird diese Entscheidung beim Eincheckvorgang der Programmversion durch den Entwickler und Administrator gemeinsam getroffen. Dabei ist möglich, daß jede Version als Hauptversion nur sich selbst definiert.

4. *Executable*

   Auf dieser Ebene finden sich die ausführbaren Dateien, deren Namen aus einer beliebigen Zeichenkette und der betreffenden Versionsnummer in eckigen Klammern (`<File>[<VersionNumber>]`) gebildet werden sowie die Ein- und Ausgabedateien, die zur Ausführung benötigt werden.

Dieses Verzeichnis ist gleichzeitig der Ort der Ausführung („*Working Directory*“), wobei die Programmversionen nur auf Dateien zugreifen dürfen, die im oder unterhalb des *Working Directory's* gelegen sind. Es darf keine Annahme darüber getroffen werden, welche Dateienstruktur sich oberhalb desselben befindet.

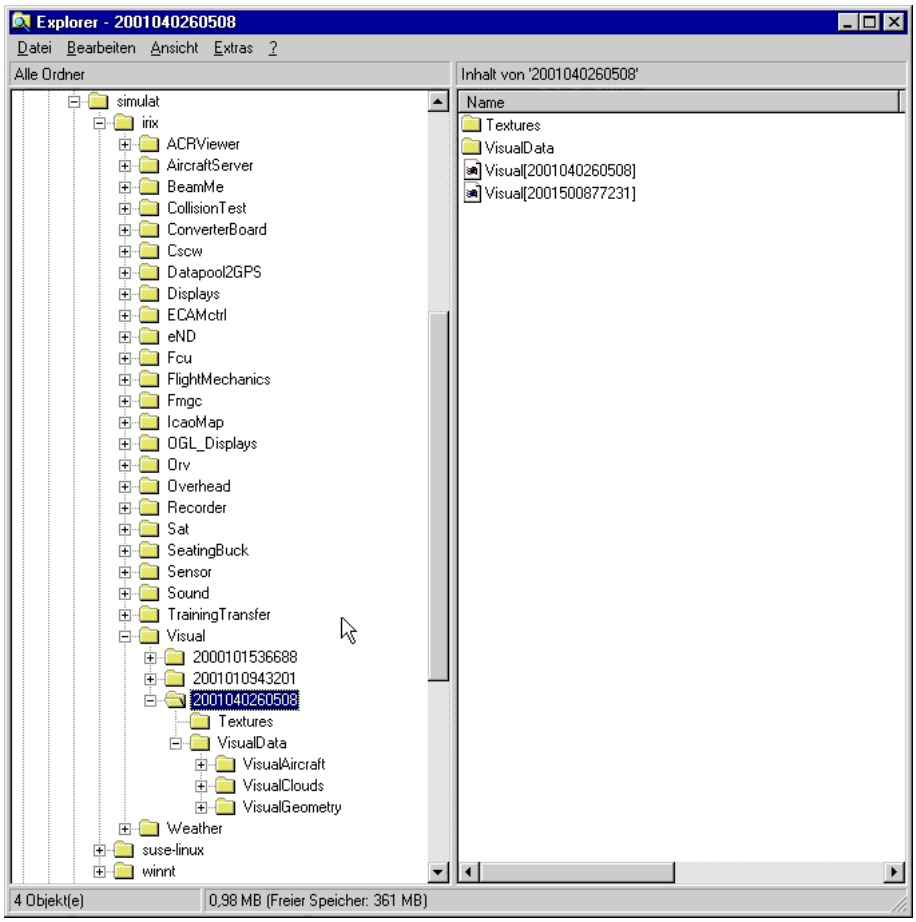


Abb. 7.13: *NeMO's Program Database* (NePD) - Beispiel

Analog zu den Simulationsprogrammen existiert auch für die Verwaltung der dynamischen Laufzeitbibliotheken (*Dynamic Link Library*) eine entsprechende Verzeichnisstruktur. Die *Dynamic Link Libraries* werden von den Programmen zur Ausführung geladen und stellen gemeinsam genutzte Funktionalitäten zur Verfügung (vergleiche auch die Ausführungen zum Programmabbild in Abschnitt 6.3.2).

Da die Laufzeitbibliotheken in unterschiedlichen Formaten für ein Betriebssystem vorliegen können (für *SGI IRIX* beispielsweise als 32-Bit- oder 64-Bit-Objekt) wird

unterhalb der Betriebssystemebene noch ein sogenannter *DllSubPath* definiert, der zwischen den Formaten unterscheidet (z.B. `lib32` und `lib64`).

Informationen darüber, wo Simulationsprogramme und Laufzeitbibliotheken von *Nemo's Model Organizer* erwartet werden, können auch jederzeit über das in Kapitel 7.1.3 vorgestellte *Monitor Widget* abgefragt werden. Dort wird aus der Sicht einer Programmversion angezeigt, wo die entsprechenden Dateien zu liegen kommen müssen (siehe Abbildung 7.14).

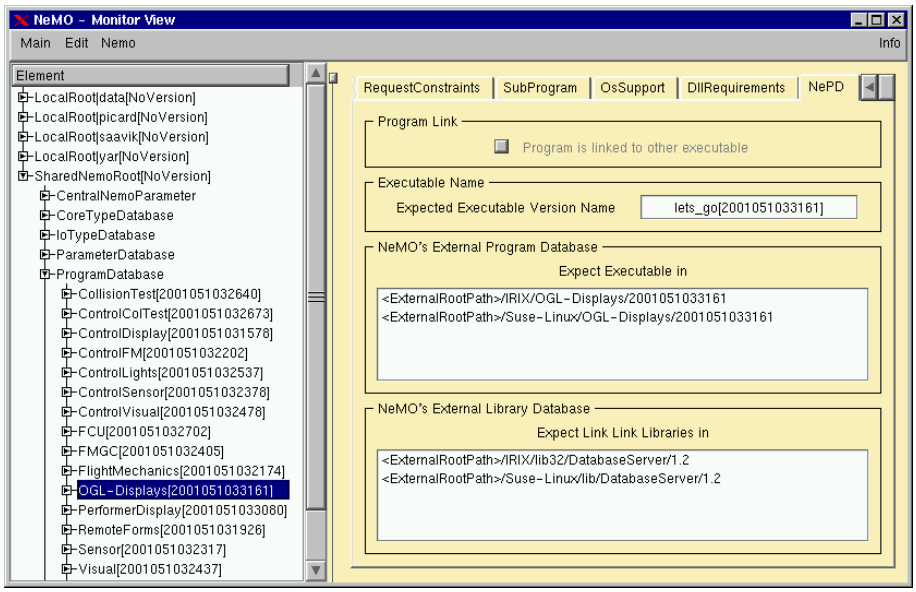


Abb. 7.14: Informationen über *NeMO's Program Database* im *Monitor Widget*

Werden die vorgestellten Randbedingungen und Voraussetzungen eingehalten, so kann *NeMO's Program Database* (NePD) durch den Administrator beliebig auf den Rechnern positioniert werden und ist dadurch unabhängig von der am Fachgebiet Flugmechanik und Regelungstechnik vorhandenen Hardware.

7.2.3 *NeMO's Communication Database*

Die Aufgabe des Modellmanagements ist unter anderem die Dokumentation des Kommunikationsverhaltens von Prozessen und Programmen, sei es zum Zwecke der Überwachung, der Analyse, etc.. In Kapitel 4.4 ist erläutert worden, daß für die Analyse der verwalteten Simulationsmodelle auch eine von den Prozessen und Programmen losgelöste Datensicht benötigt wird.

NeMO's Communication Database (NeCD) bietet diese rein inhaltliche orientierte, abstrakte Betrachtung der in der Simulationsumgebung ausgetauschten Informationen und ist dadurch losgelöst von allen Umsetzungsdetails (welcher Prozeß ist für eine Variable verantwortlich, welche Prozesse gehören inhaltlich zusammen, wie ist der Datenfluß von einem zum anderen Prozeß, ...).

Die Kommunikations-Datenbank enthält alle in der Simulationsarchitektur *DSPA* bekannten Aliase, repräsentiert also quasi die Vereinigungsmenge der Informationen aus allen Simulationsmodellen. Jeder Entwickler eines Simulationsmoduls muß die von seinem Modul produzierten Informationen mit Hilfe von *NeMO's Communication Database* registrieren und veröffentlichen, um auf diese Weise ein *Interface Control Document* (ICD) für die gesamte Simulationsumgebung zur Verfügung zu stellen.

Da die Alias-Datenbank nicht nur den Zwecken von *Nemo's Model Organizer* dient, sondern auch für das Datenmanagement und den Menschen direkt seine Bedeutung besitzt, muß die bisher eingeschränkte Definition eines Aliases (vergleiche Abschnitt 6.2.2) erweitert werden. In Ergänzung der bereits erläuterten Eigenschaften wie *Namen* und *Typ* wird ein Alias in *NeMO's Communication Database* durch folgende Information charakterisiert:

- *Size*

  Dies stellt eine Anforderung aus dem Datenmanagement dar, da für eine Übertragung eines Aliases von einem Rechner zum anderen die Größe der Variablen (Anzahl der Bytes) zur korrekten Interpretation unbedingt benötigt wird.

- *Oberbegriff*

  Ein Alias existiert in der Regel nicht ohne Bezug zu anderen, da ein Objekt oder Modell meist durch eine Menge von Informationen beschrieben wird. Welche der Aliase inhaltlich zusammengehören, wird über den Oberbegriff zum Ausdruck gebracht.

- *Beschreibung*

  Der Definition eines Aliases wird eine textliche Beschreibung hinzugefügt, die die Bedeutung desselben für den Menschen verdeutlichen soll.

- *Einheit und Wertebereich*

  Aliase können in Berechnungen eingehen. Die Informationen über die Einheit und den Wertebereich eines Aliases ist in diesem Zusammenhang von großer Bedeutung und kann helfen, eventuelle Fehler aufzuspüren oder zu vermeiden.

- *Ansprechpartner*

  Reichen die Informationen in der Kommunikations-Datenbank nicht aus, so wird durch diese Angabe eine Person benannt, die für weitere Fragen zur Verfügung steht. In der Regel handelt es sich um den Entwickler des Simulationsmoduls, welches den Alias produziert.

NeMO's Communication Database wurde schon parallel zur Entwicklung des restlichen Modellmanagements aufgebaut und benötigt, so daß für diese Datenbank auf die Software „MICROSOFT® ACCESS“ zurückgegriffen wurde.

Den bisherigen Gepflogenheiten folgend wird auch der Einsatz der Kommunikationsdatenbank über eine Benutzeroberfläche unterstützt, die in diesem Falle mit Hilfe der in der Software „ACCESS“ integrierten Funktionalitäten programmiert worden ist. Abbildung 7.15 zeigt das Hauptfenster des sogenannten *Alias Manager's*.

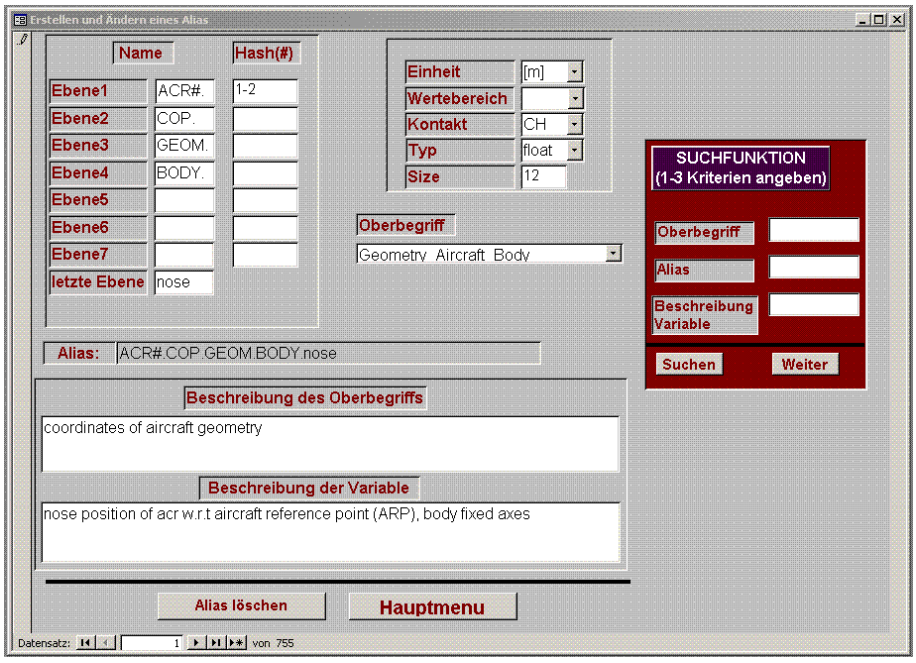


Abb. 7.15: *Alias Manager* (Main Window)

In der linken, oberen Ecke erkennt man die Zusammensetzung eines Aliases aus Ebenen, wobei der Buchstabe „#“ als Platzhalter für beliebige Zahlen dient. Bei Bedarf kann der Platzhalter auf sinnvolle Wertebelegungen eingegrenzt werden, was im jeweiligen Nachbarfeld angegeben wird. Unterhalb der einzelnen Ebenen wird der Alias in seiner Gesamtheit dargestellt.

Die anderen Kategorien wie *Einheit*, *Wertebereich*, *Kontaktperson*, *Typ* und *Größe* finden sich in der oberen Bildmitte, abgerundet von dem Eingabefenster für den *Oberbegriff*, dem dieser Alias zuzuordnen ist. Eine *Beschreibung* desselben sowie des spezifischen Alias kann in der unteren Hälfte der Benutzeroberfläche angegeben werden.

Die rechte Hälfte des Fensters enthält eine Suchmaske, mit deren Hilfe die Alias-Datenbank durchsucht werden kann. Als Suchkriterien können Informationen zu den Kategorien *Aliasname*, *Oberbegriff* und *Beschreibung* des Alias verwendet werden.

NeMO's Communication Database ist mit zwei Exportfiltern versehen worden. Der erste erlaubt die Erstellung eines für den Menschen gut lesbaren Berichts (siehe Abbildung 7.16), der beispielsweise als *Interface Control Document* an externe Partner vergeben werden kann, mit denen die verteilte Simulationsarchitektur *DSPA* gekoppelt werden soll.

Der zweite Filter erzeugt eine reine ASCII-Datei, in der die Aliase zeilenweise enthalten sind, die verschiedenen Kategorien durch Semikolon getrennt (vergleiche Abbildung 7.17).

Diese für den Menschen schwer verständliche Darstellung ist für eine automatisierte Verarbeitung durch andere Programme gedacht. Außerdem kann der Inhalt dadurch einfach auf andere Plattformen übertragen bzw. in andere Software integriert werden. Somit stellt die Auswahl von MICROSOFT® ACCESS keine wirkliche Beschränkung auf das *Windows*-Betriebssystem dar.

Zusammenfassend ist der Nutzen von *NeMO's Communication Database* dreierlei:

- Alle in der Simulation ausgetauschten Daten werden an einer zentralen, für jedermann einsichtigen Stelle festgehalten und verwaltet.
- Es wird die Möglichkeit geschaffen, eine Aussage über das Datenprofil eines Simulationsvorhabens zu treffen, beispielsweise ob eine Information unbekannt ist und damit eine potentielle Fehlerquelle darstellt oder ob ein gewünschter Alias überhaupt zur Verfügung gestellt werden kann.
- Die Alias-Datenbank enthält alle Informationen, die für den Austausch von Aliasen über das Datenmanagement benötigt werden. Deswegen ist im Rahmen der Software *Octopus* ein Codegenerator (*Octopus Box Builder* (OBB); vergleiche [Eng01]) mit der Datenbank verbunden worden, der den Entwickler bei der Programmierung des Datenaustauschs entlastet.

Oberbegriff	Name	Beschreibung	Einheit	Bereich	Kontakt	Size	Typ
Geometry_Aircraft_Body	**ACR#.COP.GEOM.BODY.nose_gear**	nose gear position of acr w.r.t. aircraft reference point (ARP), body fixed axes	[m]		CH	12	**float**
Geometry_Aircraft_Body	**ACR#.COP.GEOM.BODY.r_elev_tip**	right elevator tip position of acr w.r.t. aircraft reference point (ARP), body fixed axes	[m]		CH	12	**float**
Geometry_Aircraft_Body	**ACR#.COP.GEOM.BODY.r_main_gear**	right main gear position of acr w.r.t. aircraft reference point (ARP), body fixed axes	[m]		CH	12	**float**
Geometry_Aircraft_Body	**ACR#.COP.GEOM.BODY.r_wingtip**	right wingtip position of acr w.r.t. aircraft reference point (ARP), body fixed axes	[m]		CH	12	**float**
Geometry_Aircraft_Body	**ACR#.COP.GEOM.BODY.rudder_top**	rudder top position of acr w.r.t. aircraft reference point (ARP), body fixed axes	[m]		CH	12	**float**
Geometry_Aircraft_Body	**ACR#.COP.GEOM.BODY.tail**	tail position of acr w.r.t aircraft reference point (ARP), body fixed axes	[m]		CH	12	**float**
Geometry_Data_Body_Pilots_T	**ACR#.COP.GEOM.PILOT#.BODY.body_pos**		[]		CH	960	**float[15][4][4]**
Geometry_Data_Pilots	**ACR#.COP.GEOM.PILOT#.head_pos**	Pilot1=Captain, Pilot2=First Officer transformation matrix cockpit origin -> pilot's head	[mm]		JS	64	**float[4][4]**

Abb. 7.16: Interface Control Document - Rich Text

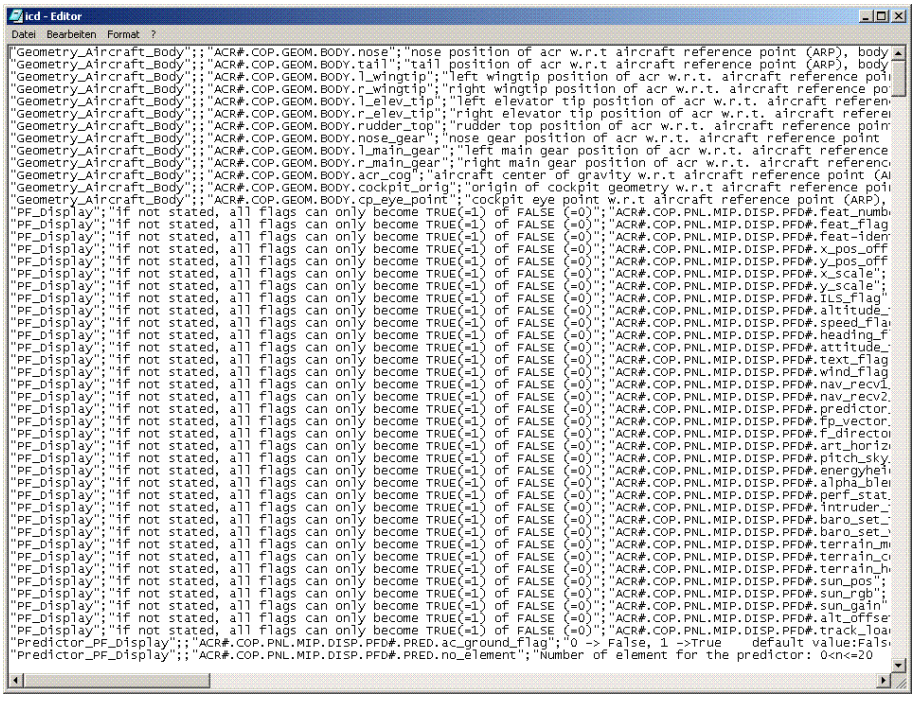


Abb. 7.17: Interface Control Document - ASCII

8 Bewertung der Aufgabe

Das Modellmanagement hat sich zur Aufgabe gemacht, in Zusammenarbeit mit dem Datenmanagement ein Rahmenwerk zu verwirklichen, welches die Entwicklung und den Einsatz von Computer-Simulationsmodellen am Fachgebiet Flugmechanik und Regelungstechnik derart unterstützt, daß ein Entwickler sich voll und ganz auf die Lösung seiner „persönlichen“ Problemstellung konzentrieren kann (*Simulation Support Environment*). „Persönlich“ heißt in diesem Zusammenhang, daß ein Entwickler in Abhängigkeit von seinen Forschungsprojekten abgetrennte und relativ einfach überschaubare Teilbereiche eines Simulationsmodells bearbeitet, adaptiert oder optimiert.

Auf der anderen Seite ist ein Entwickler auch meist Benutzer der Simulation, wobei er naturgemäß auf die gesamte Funktionalität der Simulationsumgebung zurückgreifen möchte und nicht nur auf seine eigene Entwicklung.

Die Überwindung dieses scheinbaren Gegensatzes zwischen der weitestgehend entkoppelten Arbeitsweise eines Entwicklers einerseits und dem Wunsch einer einfachen und einheitlichen Benutzung aller Teillösungen als Gesamtes andererseits ist das Ziel der verteilten Simulationsarchitektur *DSPA* mit ihren Organisationseinheiten Daten- und Modellmanagement. Dazu begleitet sie den Menschen während des gesamten Lebenszyklus eines Simulationsmodells und schließt die Lücken im Ablauf, die nicht in den Verantwortungsbereich der Gruppe der Entwickler fällt.

Dieser Abschnitt betrachtet, ob die in den beiden vorangegangen Kapiteln beschriebene Umsetzung des Modellmanagements diesen Ansprüchen gerecht wird. Dazu wird zuerst die Erfüllung von *NeMO*'s unmittelbaren Aufgaben, Integration der Simulationsprogramme in die Architektur sowie Konfiguration und Überwachung der verteilten Prozeßstruktur von Simulationsvorhaben, eingehend diskutiert werden.

Im Anschluß daran erfolgt eine Erörterung der Leistungsfähigkeit und der Möglichkeiten des zentralen Arbeitsplatzes des Modellmanagements, von dem aus der Mensch in die Simulation eingreift. Der erste Teil der Bewertung wird abgerundet durch eine Betrachtung des Datenbankkonzepts. Zentraler Punkt ist dabei die Frage nach der Vollständigkeit der gespeicherten Daten als auch die Flexibilität hinsichtlich der verwendbaren Rechnerplattformen und möglicher zukünftiger Erweiterungen.

Der zweite Teil dieses Kapitels überprüft, ob die Realisierung von *Nemo's Model Organizer* nicht gegen die zentralen Leitlinien der *DSPA* (modular, verteilt, flexibel, skalierbar) verstößt bzw. sie nur unzureichend berücksichtigt.

8.1 Teilaufgaben

Integration

Nemo's Model Organizer muß die Lücke schließen zwischen der abgetrennten, relativ isolierten Welt des Entwicklers einer einzelnen Komponente und dem Zusammenwirken aller Module in der gesamten Simulation. Dies wird im wesentlichen erreicht durch die Definition eines Komponentenmodells für die Simulationsmodule in der verteilten Simulationsarchitektur *DSPA* (vergleiche Kapitel 6.3).

Komponenten sind in der Theorie der Softwareentwicklung eigenständige Einheiten, die zu einer meist verteilten Anwendung zusammengesetzt werden können. Dem Aspekt der Kombinierbarkeit und Integrierbarkeit von Komponenten wird in diesem Zusammenhang besondere Aufmerksamkeit geschenkt. Demzufolge sind die Schwerpunkte von *NeMO*'s Komponentenmodell die Beziehungen der Komponenten untereinander, die Bedienung von Modulen sowie die von den Komponenten benötigte Laufzeitumgebung.

Die Beziehungen von Komponenten werden aufgrund des sogenannten Kommunikationsverhaltens von Prozessen und Programmen erarbeitet und bewertet. Das Verhalten stellt zugleich die Basis für den in dieser Arbeit entwickelten Signaturbegriff dar, so daß eine Identifikation und Charakterisierung von Prozessen und Programmen erst möglich wird. Diese Vorgehensweise bietet den Vorteil, daß das Modellmanagement vollständig unabhängig vom Simulationsmodell agieren kann, da außer dem Vorhandensein des Datenmanagements und dessen Funktionalität nichts von *Nemo's Model Organizer* vorausgesetzt wird. Diese vermeintliche Einschränkung ist in Wahrheit keine, da die Simulationsprozesse selbst das Datenmanagement für die Erfüllung ihrer Aufgaben benötigen.

Durch die Berücksichtigung der Bedienung eines Simulationsmoduls als Bestandteil der Modellierung wird das Modellmanagement in die Lage versetzt, eine Gebrauchsanweisung zu erstellen, nicht nur für eigene Zwecke, sondern auch zur Anleitung des Benutzers.

Die Mechanismen, die zur Einstellung eines Prozesses verwendet werden dürfen, sind entweder allgemeingültig auf jeder Betriebsplattform einsetzbar (Kommandozeilenparameter, Prozeß-Environment) oder sie benutzen nach einem vorgegeben Muster das Datenmanagement zur Informationsübertragung an den Prozeß. Letzteres meint die Aufteilung eines Simulationsmoduls in Ausführungs- und Kontrollpart, wobei der

Kontrollpart in der Regel eine graphische Benutzeroberfläche enthält, mit deren Hilfe der Benutzer der Simulation Kommandos an den Ausführungsteil senden kann. Durch die Beschränkung auf die genannten Mechanismen wird die Unabhängigkeit des Modellmanagements von betriebssystemspezifischen Aspekten gewahrt.

Wesentlich für den Einsatz von Komponenten als Teil eines Simulationsvorhabens ist die Bereitstellung einer Laufzeitumgebung, die auf den Simulationsprozeß abgestimmt ist. Nur wenn ein Prozeß die Ressourcen zugeteilt bekommt, die er zur Ausführung braucht, kann er seine inhaltliche (d.h. in Bezug auf das Simulationsmodell) Aufgabe zufriedenstellend ausfüllen. Voraussetzung für eine entsprechende Formulierung und Umsetzung der Anforderungen von Prozessen ist das in Abschnitt 6.1 vorgestellte Rechnermodell.

Die Definition und Vorgabe einer Integrationsprozedur unter dem Aspekt der Qualitätssicherung ergibt sich dann fast zwangsläufig aus der Summe der Informationen, die für obige Modellierung benötigt wird. Eine entsprechende Spezifikation darüber, welche Informationen wie und in welcher Reihenfolge gewonnen werden müssen, ist in Abschnitt 6.3.4 im Detail festgehalten. Der Vorgang ist semi-automatisiert, insofern der Entwickler eines Simulationsmoduls als Informationsquelle vorausgesetzt wird. Seine „Belastung“ beschränkt sich aber auf die Beantwortung von Fragen mit Hilfe einer graphischen Benutzeroberfläche (siehe Kapitel 7.1.4).

Der Vorgang der Integration läßt, wie in Abschnitt 5.5 gefordert, die Simulationsmodule unangetastet und produziert keinerlei Seiteneffekte im Hinblick darauf, daß ein Simulationsprogramm nur noch in Kombination mit dem Modellmanagement zum Einsatz kommen kann. In einem solchen Falle könnten Simulationsprogramme beispielsweise nicht mehr zur Erprobung in Forschungsprojekten bei externen Partnern verwendet werden, was eine nicht unerhebliche Einschränkung der Zusammenarbeit mit anderen Instituten bedeuten kann.

Deswegen besteht der Schritt der Integration aus einer reinen Sammlung von Informationen, die später zur Beobachtung der Komponenten eingesetzt wird. Außerdem wird die Freiheit der Entwickler in der Umsetzung ihrer Simulationsprogramme so wenig wie möglich eingeschränkt. Selbst eine Nichtbeachtung der in Kapitel 6.3.3 formulierten Richtlinien führt nicht zum sofortigen Ausschluß eines Simulationsprogrammes.

Konfiguration

Das Modellmanagement hat die Aufgabe, die Kombination der Teillösungen der Entwickler zu einem Gesamten zu unterstützen, und zwar abhängig vom jeweiligen Simulationsvorhaben.

Zu diesem Zweck greift *NeMO* auf zwei Informationsquellen zu: das Datenmanagement als „online“-Quelle und der Integrationsvorgang, mit dessen Hilfe Wissen „offline“ gewonnen werden kann. Diese Kenntnisse sind ausreichend, um die in der Simulationsarchitektur bekannten Komponenten zu einer verteilten Anwendung zu kombinieren.

Die Anwendung des Wissens muß nicht durch den Menschen erfolgen, sondern wird weitestgehend von *Nemo's Model Organizer* übernommen. Der Mensch wird nur an bestimmten Stellen vom Modellmanagement eingebunden, wenn es darum geht eine Auswahl zu treffen, einen Sachverhalt zu bestätigen oder einen Wunsch zu äußern. Die dem Menschen gegenüber formulierten Fragen sind entweder inhaltlicher Natur, d.h. beziehen sich auf das vom Benutzer geplante Simulationsvorhaben, oder sind derart aufbereitet, daß auch ein Anwender mit geringeren Kenntnissen über die Simulationsumgebung diese beantworten kann.

Die Wünsche des Benutzers werden dann vom Modellmanagement in eine verteilte Prozeßstruktur „übersetzt“, was aufgrund der Komplexität der Tätigkeit in mehreren Teilschritten geschieht: Programmauswahl, Parameterbelegung, Ressourcenverhandlung und Vorbereitung der Ausführung.

Zur Programmauswahl kommen die in Abschnitt 7.1.2 beschriebenen Selektoren zur Verwendung, die Teilbereiche des Simulationsmodells oder Ausrüstungsgegenstände des Forschungscockpits mit Simulationsprogrammen assoziieren und damit den Hauptschritt in Richtung Prozeßstruktur tätigen. Die Einstellung der gewählten Simulationsprogramme geschieht mit der Hilfe von Parametern, deren Modellierung Bestandteil von *NeMO*'s Komponentenmodell ist. Die Tätigkeit des Nutzers beschränkt sich infolge dessen meist nur noch auf die Auswahl von Parametern aus einer Liste von Möglichkeiten, die eine textliche Beschreibung der möglichen Parameterbelegungen enthält. In manchen Fällen, wo Parameter global für alle Simulationsprozesse gelten oder Voreinstellungen verwendet werden sollen, wird der Anwender überhaupt nicht mehr gefragt.

Im dritten Schritt wird anhand von Anforderungen des Prozesses festgestellt, auf welchen Rechnern er überhaupt ausgeführt werden könnte. Basis hierfür ist wieder-

um die Modellierung von Rechnerbausteinen und Anforderungen, die an die Leistungsfähigkeit von Rechnern gestellt werden (vergleiche Kapitel 6.1). Findet sich ein Computer, der in der Lage ist, den Simulationsprozeß zu übernehmen, so werden vom Modellmanagement im Zuge einer sogenannten Vorverwaltung die entsprechenden Ressourcen für diesen Prozeß reserviert.

Als letztes wird vom Modellmanagement erwartet, daß es dem Simulationsprozeß die entsprechende Laufzeitumgebung zur Verfügung stellt. In diesem Zusammenhang sorgt *NeMO* für die Übergabe des Prozesses an das Betriebssystem, die Weiterleitung der Parameter, die Bereitstellung der vom Prozeß erwarteten Struktur von Ein- und Ausgabedateien sowie die Gewährleistung des Zugriffs auf die benötigten Laufzeitbibliotheken. Von diesem Augenblick an steht der Simulationsprozeß für das Simulationsvorhaben zur Verfügung.

Die den einzelnen Schritten zugrundeliegenden, unterschiedlichen Betrachtungsebenen (Ressourcen, einzelner Prozeß, Prozeßstruktur, etc.) werden durch das Modellmanagement in einen Gesamtzusammenhang gebracht und dem Menschen über einen zentralen Arbeitsplatz zugänglich gemacht.

Die Zusammenstellung der Module ist nicht a priori beschränkt, sondern es kann prinzipiell jede Komponente mit allen anderen kombiniert werden. In dieser Hinsicht bietet *NeMO* die maximal mögliche Flexibilität, was implizit durch die Rahmenbedingungen der Simulationsarchitektur vom Modellmanagement gefordert wird.

Die „stufenlose“ Kombination von Simulationsprogrammen zur Ausführung in einer Simulation soll aber nicht jedesmal durchgeführt werden müssen. Aus diesem Grunde können Teilbereiche oder ein gesamtes Simulationsvorhaben vorkonfiguriert werden und in *NeMO's Simulation Database* (NeSD) abgespeichert werden. Besteht Bedarf das dadurch charakterisierte Simulationsmodell erneut einzusetzen, kann es anhand der Daten aus der Datenbank automatisiert erzeugt werden. Im Hinblick auf die Verwendung der Simulationsumgebung durch weniger versierte Nutzer ist diese Funktionalität ein „Muß“ für das Modellmanagement.

Ebenfalls nicht zu vernachlässigen ist die integrierte Versionsverwaltung der Simulationsprogramme. Alle Aspekte, die für die Auswahl und die Nutzung von Versionen eines Simulationsprogrammes von Bedeutung sind, werden von *Nemo's Model Organizer* erfaßt und dem Benutzer zugänglich gemacht. Dieser braucht dann lediglich anhand der Dokumentation der verschiedenen Versionen die gewünschte auszuwählen.

Überwachung

Die Notwendigkeit zur Überwachung ergibt sich daraus, daß die Kombination von Simulationsmodulen nicht vor der Laufzeit feststeht und überprüft werden konnte, sondern erst dynamisch zur Laufzeit erfolgt.

Dieser Forderung kommt das Modellmanagement nach, indem es dem Menschen über einen zentralen Arbeitsplatz einen Beobachter an die Hand gibt, der alle gesammelten Informationen aus einer ganzheitlichen Sicht heraus aufbereitet und dem Nutzer als Feedback zur Verfügung stellt. Analog zur Konfiguration wird auch die Überwachung in vier, vom Inhalt unterschiedliche Ebenen aufgeteilt: Arbeitsmittel, einzelner Prozeß, Prozeßstruktur und Datenstruktur.

Die Überwachung der Arbeitsmittel schließt Aussagen über die benutzten Rechner ein, jedoch nur aus der Benutzersicht. Falsch verkabelte Hardware im Forschungscockpit beispielsweise kann nicht vom Modellmanagement erkannt werden. Die Rückmeldungen enthalten eher Informationen, ob ein Rechner betriebsbereit ist und wie stark er ausgelastet wird. Dabei wird unter anderem wieder auf das von *NeMO* definierte Rechnermodell zurückgegriffen (siehe Kapitel 6.1).

Zu den Arbeitsmitteln gehört auch das Modellmanagement selbst. So ist die Überwachung der *Local* und *Shared Nemo* (vergleiche Abschnitt 6.4.2) unmittelbare Aufgabe von *Nemo's Model Organizer*, da nur mit deren Hilfe das Modellmanagement in der Lage ist, einerseits auf die Rechner im Netzwerk zuzugreifen und andererseits Informationen von *NeMO's Management Database* zu erfragen. Als einzig dafür in Frage kommend wird diese Anforderung durch *Central Nemo* umgesetzt (siehe Ausführungen über den *Startup Wizard*; Kapitel 7.1.1).

Auf der Ebene der Prozeßbetrachtung überprüft das Modellmanagement beispielsweise, ob ein Prozeß noch ordnungsgemäß ausgeführt wird und ob die Laufzeitumgebung den Anforderungen entspricht. Eine Stufe höher werden die Datenflüsse zwischen den Prozessen überwacht, um Kollisionen oder Unterbrechungen des Datenflusses aufzuspüren. Ein Vergleich zwischen Ist- und Soll-Verhalten kann ebenfalls vom Modellmanagement durchgeführt werden. Ersteres wird dabei durch Abfrage des Datenmanagements bestimmt, welches immer das aktuelle reale Kommunikationsverhalten eines Prozesses festhält. Letzteres stellt eine Information aus dem Eincheckvorgang dar, als Ergebnis der Analyse eines früher aufgezeichneten Kommunikationsverhaltens des Prozesses. Da während eines Simulationsvorhabens nicht immer die Zeit für eine ausführliche Analyse zur Verfügung steht, muß und wird

vom Modellmanagement die Möglichkeit geboten, alle Informationen eines Simulationsvorhabens (Prozesse und ausgetauschten Daten) abzuspeichern, um sie später in aller Ruhe begutachten zu können.

In diesem Zusammenhang ist auch *NeMO's Communication Database* (Abschnitt 7.2.3) zu erwähnen. Durch sie wird die Simulation aus einer reinen Datensicht beschrieben, was für die Analyse bestehender oder die Entwicklung neuer Module einen besseren Überblick gewährt als die Aufteilung in die komplexe Prozeßstruktur.

Anhand der Ausführung kann man erkennen, daß das Modellmanagement eine vollständige Transparenz über seine Informationen und Tätigkeiten bietet. Es gibt keine dem Menschen nicht zugänglichen Teilbereiche. Aufgrund der unterschiedlichen Wissensstände kann ein Benutzer durch die Fülle der Information leicht überfordert werden. Deswegen wird dem Nutzer die Möglichkeit geboten, eine seinen Kenntnissen angepaßte Form der Überwachung zu wählen (überblicksartig oder detailliert).

Die Überwachung durch *Nemo's Model Organizer* ist teilweise automatisiert, so daß eine Fehlerbehebung ohne den Menschen erfolgen kann. Dies kann jedoch nicht für alle denkbaren Fehler garantiert werden, so daß das Minimalziel darin bestehen muß, den Menschen auf Fehlerfälle hinzuweisen und ihn dahingehend zu unterstützen, den Fehler in seinen Kontext einzuordnen. Dadurch wird er in die Lage versetzt, zu beurteilen, worauf sich ein Fehler auswirken kann, wer in Mitleidenschaft gezogen wird und wie hoch die Bedeutung für das Gesamtsystem ist. Ein anschauliches Beispiel ist die Information, welche Cockpitanzeige durch einen fehlerhaften Simulationsprozess getrieben wird oder wer der Super-Prozeß eines ausgefallenen Sub-Prozesses ist.

Die bisher beschriebene Form der Überwachung ist immer davon ausgegangen, daß das Modellmanagement erst aktiv wird, wenn ein Mißstand aufgetreten ist. Alternativ versucht *Nemo's Model Organizer* jedoch Fehler vorherzusehen und präventiv zu handeln, d.h. sie entweder zu vermeiden oder den Menschen darauf hinzuweisen. Dazu gehört zuallererst die Vorhersage des Kommunikationsverhaltens von Simulationsprozessen (Kapitel 6.2.3). Andere Beispiele hierfür sind die Sperrung von Rechnern, deren Ressourcen bereits von plazierten Prozessen ausgelastet sind, oder die Nichtbeachtung globaler Parameter.

Zusätzlich zur durch das Modellmanagement initiierten Überwachung kann der Mensch jederzeit Eigeninitiative entwickeln und über das *Monitor Widget* (siehe Abschnitt 7.1.3) alle *NeMO* zur Verfügung stehenden Informationen einsehen.

Abschließend ist die Überwachung durch das Modellmanagement nicht starr konzi-

piert, sondern kann anhand neuer Informationen jederzeit adaptiert werden. Dazu zählt die Überprüfung des Kommunikationsverhaltens von Programmen, welches möglicherweise fehlerhaft bestimmt worden ist (vergleiche Kapitel 6.2.4). In diesem Fall wird entweder durch das Modellmanagement eine erneute Analyse durchgeführt oder die Informationen über den Prozeß können „von Hand“ editiert werden.

Bedienoberfläche

Das Modellmanagement soll ein Hilfsmittel für den Menschen sein, so daß er in die Lage versetzt wird, das von ihm beabsichtigte Simulationsvorhaben auch durchzuführen. Demzufolge braucht er einen Zugang zu der verteilten Anwendung *Nemo's Model Organizer*.

NeMO stellt dem Menschen einen zentralen Arbeitsplatz in Form einer graphischen Benutzeroberfläche zur Verfügung (siehe Kapitel 7.1), weil eine dezentrale und kommandobasierte Verwaltung der Simulationsumgebung die Fähigkeiten eines „normalen“ Benutzers in der Regel überfordert. Die Komplexität der Aufgabe wäre nur noch von Experten beherrschbar, was den Zielen des Modellmanagements widerspricht.

Die Forderung nach einem zentralen Arbeitsplatz beinhaltet, daß die Benutzeroberfläche eine integrale Lösung darstellen muß, über die alle Tätigkeiten im Zuge der Verwaltung der Simulationsmodelle ausgeführt werden können. *NeMO* entspricht dieser Anforderung, was sich unter anderem daran erkennen läßt, daß die oben beschriebenen Aufgaben Integration, Konfiguration und Überwachung die Gliederung der Oberfläche maßgeblich bestimmt haben:

- Integration → *Checkin Wizard*
- Konfiguration → *Model Setup Widget* (primär)
- Überwachung → *Monitor Widget* (primär)

Abgerundet wird der Umfang der graphischen Bedienoberfläche durch einen Anteil zum geordneten Hochfahren des Modellmanagements (*Startup Wizard*).

Das oben schon angedeutete, unterschiedliche Klientel des Modellmanagements (Benutzer, Entwickler und Administrator; vergleiche Abschnitt 3.2) wird ebenfalls in der Gliederung und Gestaltung der graphischen Bedienoberfläche berücksichtigt.

Im Normalfall versucht *Nemo's Model Organizer* den Menschen so weit wie möglich anzuleiten und erwartet keine tiefergehenden Kenntnisse vom Menschen (Vorstellung

eines Standardnutzers; *Model Setup Widget*). Der Administrator und gegebenenfalls die Entwickler können hingegen ihrem größeren Wissensstand entsprechend in die „Tiefen“ des Modellmanagements Einblick nehmen (*Monitor Widget*). Allgemein versucht das Modellmanagement, die Komplexität der Darstellung gering zu halten und immer nur die momentan benötigten Informationen anzuzeigen. Von einer Übersichtsdarstellung kann dann immer auf eine Detailsicht gewechselt werden.

Der vollständige Zugriff auf alle Objekte des Modellmanagements, ohne deren Funktionalität zu beeinträchtigen, wird dadurch erreicht, daß die Benutzeroberfläche vollständig parallel zum Rest von *Nemo's Model Organizer* aufgebaut worden ist. In diesem Zusammenhang ist die Definition eines Objekts um die Festlegung einer graphischen Repräsentation erweitert worden, so daß das Objekt auch selbst vorschreibt, wie die Benutzeroberfläche auf es zugreifen kann. Die Umsetzung der Oberfläche kann dann getrennt vom eigentlichen Objekt erfolgen.

Der verwendetete Kommunikationsmechanismus (CORBA) ist derselbe wie für die Aktion der Objekte untereinander. Unter diesem Gesichtspunkt kann man die Benutzerschnittstelle als ein weiteres Objekt in der Gesamtheit des Modellmanagements betrachten, mit dessen Hilfe der Mensch in die Objektstruktur integriert wird und auf sie Einfluß nehmen kann.

Abschließend sei hervorgehoben, daß die graphische Bedienoberfläche nicht statisch und unveränderbar ist. Beispielsweise hinsichtlich der Gestaltung des *Model Setup Widget's* kann der Mensch interaktiv und dynamisch die Oberflächen an seine Wünsche und Vorstellungen anpassen.

Datenbankanbindung

Nemo's Model Organizer benötigt zur Erfüllung seiner Aufgaben ein „Gedächtnis“, da nicht alle Informationen zur Laufzeit gewonnen oder vom Benutzer neu abgefragt werden können. Aus diesem Grund wird das Modellmanagement an eine Datenbank angebunden (*NeMO's Simulation Database*; siehe Kapitel 7.2). Zusätzlich kann auf diese Weise das Wissen über die Simulation am Fachgebiet Flugmechanik und Regelungstechnik für nachfolgende Generationen erhalten werden.

Der Umfang der gespeicherten Daten bildet nicht nur einen Teil des Modellmanagements, sondern bis auf einige wenige, nur zur Laufzeit interessierenden Informationen den gesamten Zustand von *Nemo's Model Organizer* ab: Arbeitsmittel, Simulationsmodelle, Eigenschaften der Komponenten, die Simulationsprogramme, Session-Logs

der Prozesse und Daten eines Simulationsvorhabens, Benutzerschnittstelle, globale Einstellungen des Modellmanagements, Datenbaum, etc..

Der unterschiedlichen Natur und Verwendung der Daten wird das Modellmanagement gerecht durch eine Aufteilung in drei unabhängige Teildatenbanken (*NeMO's Management*, *Program* und *Communication Database*). Diese können jeweils mehrfach vorliegen, wobei sie anhand ihres Namens und ihres Ortes unterschieden werden. Auf diese Weise ist eine Verwaltung verschiedener Umgebungen durch das Modellmanagement möglich. Ein Wechsel beispielsweise auf eine Programm-Datenbank für eine andere Art von Simulation ließe sich durch Laden einer Datei (*Management Database*) und durch die Angabe eines Verzeichnisses (*Program Database*) einfach und schnell bewerkstelligen.

Aufgrund der Aufteilung der Datenbank bietet sich die dezentrale Verwaltung der Informationen an. Das Modellmanagement setzt dies dadurch um, daß Informationen von lokaler Bedeutung dem Verantwortungsbereich der *Local Nemo* und gemeinsam genutzte Daten dem Verantwortungsbereich von *Shared Nemo* zugeordnet werden. Intern wird die Dezentralisierung sogar bis auf Objektebene heruntergebrochen (vergleiche Abschnitt 6.4.1), so daß hinsichtlich der Speicherung von Information die Flexibilität des Modellmanagements sehr groß ist.

Für die Bereiche der Simulationsdatenbanken, die mit Hilfe des Administrators verwaltet werden müssen, gibt *NeMO* Hilfe und Anleitung, da die hohe Flexibilität und die Aufteilung der Datenbank in mehrere kleine Einheiten sonst leicht unübersichtlich werden kann.

Die Daten in allen Datenbanken sind einer Begutachtung durch den Menschen zugänglich, d.h. sie sind lesbar und entsprechend ihrer Struktur im Modellmanagement angeordnet. Beispielsweise sind die Informationen in der *Management Database* in der gleichen Baumhierarchie abgelegt, wie sie zur Laufzeit existieren. Durch die Möglichkeit der *Offline*-Begutachtung der Datenbanken wird die Transparenz erhöht und somit ein besseres Verständnis von *Nemo's Model Organizer* ermöglicht.

Abschließend soll hervorgehoben werden, daß die Anbindung des Modellmanagements an eine Datenbank keine Beschränkung auf eine spezifische Betriebsplattform und auch keine Abhängigkeiten von einer bestimmten Software (außer dem Modellmanagement selbst) nach sich zieht. Dies wird beispielsweise durch die Nutzung von standardisierten Ein- und Ausgabemechanismen aus der *C++*-Laufzeitbibliothek erreicht.

Die Verwendung der Software MICROSOFT® ACCESS für *NeMO's Communicati-*

on Database ist zwar nur auf dem Betriebssystem *Windows* möglich, es sind jedoch Möglichkeiten vorhanden, wie die in der Datenbank enthaltenen Informationen auf andere Plattformen portiert werden können. Dementsprechend ist die Kommunikations-Datenbank eher als Relikt aus der Entwicklungszeit des Modellmanagements zu sehen, als dieses noch nicht einsatzbereit war, und nach einer praktikablen Zwischenlösung gesucht werden mußte.

8.2 Rahmenbedingungen

Das Modellmanagement ist Bestandteil der verteilten Simulationsarchitektur *DSPA* und dieser untergeordnet. Aus diesem Grund müssen die Richtlinien der Simulationsarchitektur (siehe 2.2.2) beachtet werden: **modular**, **verteilt**, **flexibel** und **skalierbar**.

Modular

Der Forderung nach Modularität wird *Nemo's Model Organizer* nach außen und nach innen gerecht. Im ersteren Falle werden seitens des Modellmanagements keine Vorschriften erlassen, inwiefern eine Modularisierung der Aufgaben eines Simulationsmodells zu erfolgen hat. Der oder die Entwickler haben völlige Handlungsfreiheit dahingehend, ob ein Simulationsmodell mit Hilfe eines monolithischen Prozesses oder mit einer Vielzahl kleinster Prozesse umgesetzt wird. Genauso wie die Unabhängigkeit des Modellmanagement von allen Belangen, die den Inhalt des Simulationsmodells betreffen, hat auch der Grad der Granularität in der Umsetzung keinen Einfluß auf *NeMO*.

In bezug auf die Interna des Modellmanagements wird ebenfalls strikt nach Aufgabenbereichen getrennt. Auf der obersten Ebene sind da die Rechner, die Programme bzw. Prozesse als auch die Benutzerschnittstelle zu nennen. Die tieferen Ebenen der Baumstruktur des Modellmanagements sind teilweise Abbilder der realen Prozeßstruktur eines Simulationsvorhabens und demzufolge ebenso klar nach Aufgaben unterteilt.

Allgemein kann gesagt werden, daß eine modulare Struktur den Prinzipien der Objekt-Orientierung folgt, welche für die Entwicklung des Modellmanagements konsequent angewendet worden sind.

Verteilt

Der Wunsch nach einer Unterstützung hinsichtlich verteilter Anwendungen bedeutet, daß die Simulation nicht auf einem einzelnen Rechner ausgeführt werden soll, sondern auf eine beliebige Anzahl von unterschiedlichen Rechnern zurückgreift. Dadurch daß das Modellmanagement grundsätzlich davon ausgeht, daß es sich bei einer Simulation um eine verteilte Anwendung handelt, stellt diese Anforderung kein Problem für *Nemo's Model Organizer* dar. Der Spezialfall einer Simulation bestehend aus einem Prozeß wird automatisch mit abgedeckt.

Zusätzlich macht das Modellmanagement auch keine Voraussetzung über die Art und Anzahl der Rechner, die zur Simulation verwendet werden sollen, indem es die Verwaltung der Ressourcen auf einem generischen Rechnermodell aufbaut. Auf diese Weise können die unterschiedlichsten Computer einheitlich erfaßt und behandelt werden.

Vor dem Hintergrund, daß eine verteilte Anwendung wesentlich komplexer und dadurch schlechter überschaubar ist, gibt das Modellmanagement dem Benutzer die Möglichkeit, die verteilte Prozeßstruktur von einem zentralen Arbeitsplatz aus zu überwachen. Damit fördert *NeMO* eher den Einsatz verteilter Anwendungen, als daß es diesen verhindert.

Analog zu der Forderung nach Modularität wird auch die Anforderung der Verteilung nicht nur auf die verwalteten Simulationsprogramme und Rechner bezogen, sondern auch auf den „inneren“ Aufbau des Modellmanagements. Dementsprechend sind die drei eigenständigen Einheiten *Local*, *Shared* und *Central Nemo* über das gesamte Rechnernetzwerk verteilt und dort „vor Ort“, wo sie gebraucht werden. Dabei wird auf die Kommunikationsstruktur CORBA zurückgegriffen, die eigens für verteilte Anwendungen konzipiert worden ist.

Flexibel

Flexibilität bedeutet für das Modellmanagement, so weit wie möglich allen erdenklichen Simulationsvorhaben gewachsen zu sein. In der Umsetzung von *Nemo's Model Organizer* bedeutet dies, daß der Kombinierbarkeit von Komponenten keine Grenzen gesetzt werden, die nicht in Bezug zu den modellspezifischen Inhalten stehen. Prinzipiell kann jedes Simulationsprogramm mit allen anderen zur Ausführung gebracht werden.

Daß für die unterschiedlichen Simulationsvorhaben auch unterschiedliche Benutzerschnittstellen sinnvoll sind, wird dadurch berücksichtigt, daß die Benutzeroberfläche, in diesem Falle das *Model Setup Widget*, dynamisch und interaktiv an die Gegebenheiten der Simulationsumgebung angepaßt werden kann. *NeMO* kann somit für den Festsitz-Simulator als auch für eine VR-Simulation („Virtual Reality“) am Fachgebiet Flugmechanik und Regelungstechnik zum Einsatz kommen.

In Erweiterung dieser Art von Flexibilität können die Einstellungen für eine bestimmte Simulationsumgebung in einer Datenbank abgespeichert und bei Bedarf wieder abgerufen werden. Auf diese Weise können verschiedene Konfigurationen von Simulationsumgebungen am Institut vordefiniert werden, zwischen denen ein schnelles und einfaches Wechseln ermöglicht wird.

Der Ausbau der Funktionalität der Simulation durch die Integration neuer Simulationsprogramme oder neuer Rechner stellt ebenfalls keine Schwierigkeit für das Modellmanagement dar. Ersteres wird durch die Spezifikation des Eincheckvorgangs abgedeckt, der damit den Weg ebnet für eine sichere, schnelle und trotzdem einfache Übergabe eines Simulationsprogrammes an *Nemo's Model Organizer*. Der zweite Fall bedeutet, daß der neue Rechner *NeMO*'s Rechnermodell entsprechend beschrieben werden muß. Diese Informationen werden dann an einen auf dem Rechner gestarteten *Local Nemo* übergeben, womit die Integration in das Modellmanagement abgeschlossen ist.

Flexibilität heißt zu guter Letzt auch, daß *NeMO* trotz der Bestrebung zur Automatisierung, dem Menschen immer noch die Eingriffsmöglichkeiten bietet, alles „von Hand“ nach eigenem Gutdünken einzustellen und auszuführen.

Skalierbar

Die Interpretation des Begriffes „Skalierbarkeit“ aus der Sicht des Modellmanagements bedeutet, daß die „Größenordnung“ der Simulationsmodelle beliebig vom Benutzer verändert werden kann. Konkret heißt das, daß für ein Simulationsvorhaben immer nur die Anzahl von Prozessen gestartet werden muß, die tatsächlich benötigt werden. Auf der anderen Seite kann das auch bedeuten, daß zu einer bestehenden Simulationskonfiguration aus aktuellem Anlaß noch andere Simulationsprozesse hinzugenommen werden können, ohne daß dadurch das Modellmanagement in Schwierigkeiten gebracht wird.

Gleiches gilt analog für die Verwendung der Rechner in der Simulation. Steht ein

Rechner aus irgendwelchen Gründen nicht zur Verfügung, kann über das Modellmanagement ein anderer Rechner ausgewählt werden, um die entsprechenden Simulationsprozesse auszuführen. Dabei ist es durchaus möglich die Leistungsfähigkeit eines Rechners durch die von mehreren anderen zu ersetzen.

Prinzipiell werden der Anzahl der Prozesse und der Rechner keine Grenzen durch *Nemo's Model Organizer* gesetzt. Praktische Grenzen infolge von Performance-Einbrüchen aufgrund der hohen Anzahl sind eher aufgrund der Simulationsprozesse selbst, der Leistungsfähigkeit der Rechner oder der Leistungsfähigkeit des Datenmanagements zu erwarten. Da zur eigentlichen Durchführung der Simulation die meiste Arbeit des Modellmanagements getan ist, wären selbst im Falle einer Überlastung des Modellmanagements keine großen Auswirkungen auf das Simulationsvorhaben zu erwarten. Einzig eine geringere Akzeptanz seitens des Benutzers aufgrund hoher Ansprechzeiten bei der Bedienung von *NeMO* wären zu befürchten.

Eine Überlastung des Modellmanagements ist aber eher unwahrscheinlich, da es auf der CORBA-Kommunikationsstruktur basiert, welche für den Einsatz in Projekten konzeptioniert worden ist, die die Größenordnung der Simulation am Fachgebiet Flugmechanik und Regelungstechnik deutlich überschreiten können. Außerdem sind zur Zeit Bestrebungen im Gange, die den Nutzern der *Common Object Request Broker Architecture* Möglichkeiten an die Hand geben, die Qualität der Dienstleistung („Quality of Service“) explizit festzulegen, so daß für die Zukunft Platz für eine Optimierung wäre.

8.3 Ergebnis

Anhand obiger Ausführungen kann festgehalten werden, daß *Nemo's Model Organizer* alle vorgegebenen Aufgaben in zufriedenstellender Weise erfüllt:

- Die Entwicklung von Komponenten für die Simulationsumgebung am Fachgebiet Flugmechanik und Regelungstechnik kann weitestgehend entkoppelt voneinander stattfinden.
- Trotzdem besteht mit der Hilfe des Modellmanagements die Möglichkeit, alle Simulationsmodule in einer integrierten Umgebung frei miteinander zu kombinieren und dadurch für jedes Forschungsprojekt ein maßgeschneidertes Simulationsmodell bereitzustellen.

- Die Kenntnisse, die von einem Benutzer der Simulation erwartet werden, beziehen sich hauptsächlich auf die Inhalte des Simulationsvorhabens.
- Abgerundet wird die Dienstleistung des Modellmanagements dadurch, daß eine gleichzeitige Überwachung der Simulationssession durchgeführt und der Benutzer auf auftretende Fehler hingewiesen wird.

9 Zusammenfassung und Ausblick

Die Mensch-Maschine-Schnittstelle Cockpit bietet heutzutage das größte Verbesserungspotential zur Optimierung des Luftverkehrs in bezug auf Sicherheit, Leistungsfähigkeit und Wirtschaftlichkeit. Als Forschungs- und Entwicklungsplattform wurde dazu am Fachgebiet Flugmechanik und Regelungstechnik der Technischen Universität Darmstadt eine Simulationsumgebung und ein generisches Verkehrsflugzeug-Cockpit aufgebaut.

Zur Unterstützung des Menschen bei der Entwicklung und Erstellung von Simulationsmodellen wurde eine allgemeine Struktur der Simulationsumgebung vorgegeben, die als *Distributed Simulation Programming Architecture* (*DSPA*) bezeichnet wird. Zusätzlich werden grundlegende und immer wieder benötigte Funktionalitäten über ein zweiteiliges Rahmenwerk zur Verfügung gestellt: das Datenmanagement organisiert den Datenaustausch aller Prozesse in der Simulation, während das Modellmanagement die Summe der Simulationsprozesse verwaltet, durch die ein Simulationsmodell repräsentiert wird.

Das Modellmanagement als hauptsächlicher Bestandteil dieser Arbeit hat es sich zur Aufgabe gemacht, die Lücke zwischen Entwickler einzelner Simulationsmodule und dem Benutzer der Simulation zu schließen. Dazu müssen die weitestgehend entkoppelt voneinander entwickelten Module zu einem Ganzen zusammengeführt und dem Benutzer seiner Sicht entsprechend präsentiert werden.

Die Vermittlertätigkeit umfaßt zuallererst die Überprüfung der Kooperationsfähigkeit der Komponenten untereinander sowie die Gewinnung der für eine Kooperation benötigten Informationen. Der zweite Schritt beinhaltet die Anwendung des gewonnenen Wissens, so daß der Benutzer trotz möglicher Unkenntnis über Implementationsdetails und das Zusammenspiel der Komponenten in die Lage versetzt wird, eine für sein Vorhaben maßgeschneiderte Simulation zu konfigurieren. Schließlich muß das Modellmanagement die verteilte Struktur von Simulationsprozessen zur Ausführung überwachen und überprüfen, da die individuelle Konfiguration erst dann bekannt ist und begutachtet werden kann.

Ansatzpunkt für eine Umsetzung der Anforderungen war die Analyse des Simulationsmodells bzw. der Simulationsprozesse, die das Modell auf dem Computer implementieren. Deren Verhalten und Beziehungen untereinander müssen nachvollzogen und verstanden werden, bevor sie in ihrer Gesamtheit verwaltet werden können.

Aus der Analyse ergab sich die Notwendigkeit, die in der Simulation eingesetzten

Rechner zu modellieren sowie einen Signaturbegriff für die Simulationsprozesse und -programme einzuführen. Durch ersteres werden die zur Verfügung stehenden Ressourcen und deren Leistungsfähigkeit erfaßt, so daß bezüglich der Simulationsprozesse Anforderungen formuliert werden können. Der Signaturbegriff andererseits erlaubt eine Betrachtung von Prozessen und Programmen unabhängig von ihrer inhaltlichen Simulationsaufgabe und bietet dadurch die Möglichkeit zur allgemeingültigen Identifikation und Beurteilung der Komponenten und ihres Verhaltens.

Darauf aufbauend wurde als zentraler Bestandteil des Modellmanagements ein Komponentenmodell vorgestellt. Dieses bietet die Grundlage zur Verwaltung der Prozesse und Programme, indem es diejenigen Aspekte abbildet, die für die Integration von Modulen in die Simulationsarchitektur, für die Zusammenstellung von Komponenten zu einer ausführbaren Simulation sowie für die Überwachung der verteilten Prozeßstruktur benötigt werden. Erst mit Hilfe dieser Informationen ist das Modellmanagement in der Lage, seine Aufgaben wahrzunehmen.

Im Gegensatz zu gängigen Komponentenmodellen ist dieses seiteneffektfrei in der Hinsicht, daß es die Simulationsprogramme völlig unangetastet läßt und rein aus der Beobachtersicht heraus agiert. Aus diesem Grunde können die Programme auch ohne das Modellmanagement zum Einsatz kommen, was aber aufgrund der Komplexität der Sache die Benutzung der Simulation erschweren und nur Experten zugänglich machen würde.

Anhand des Komponentenmodells wurden Richtlinien für die Entwickler formuliert, die den optimalen Nutzen des Modellmanagements gewährleisten sollen, sowie ein Eincheckvorgang spezifiziert, wie ein Simulationsprogramm in das Modellmanagement übernommen werden muß. Vorteil des Eincheckvorgangs ist die definierte Anleitung für den Menschen (Qualitätssicherung) sowie die Möglichkeit zur Automatisierung.

Die Konstruktion des eigentlichen Rahmenwerks des Modellmanagements, *Nemo's Model Organizer*, ist selbst in Form einer verteilten Anwendung realisiert worden. Auf jedem Rechner in der Simulation ist ein dem Modellmanagement zugehöriger Prozeß installiert. Damit ist quasi eine Verwaltung der Simulationskomponenten „vor Ort“ möglich. Abgerundet wird *NeMO* durch zwei zentrale Einheiten: einmal durch den Arbeitsplatz für den Menschen sowie eine Anlaufstelle für gemeinsam genutzte Informationen. Die Komponenten des Modellmanagements kooperieren mit Hilfe der verteilten Kommunikationsstruktur CORBA.

Als wesentliche Anforderung für das Modellmanagement ist ein zentraler Zugang

des Menschen formuliert worden. Dieser wird durch *Nemo's Model Organizer* in Form einer graphischen Bedienoberfläche zur Verfügung gestellt, wobei die unterschiedlichen Aufgabenbereiche sich in der Aufteilung derselben wiederfinden lassen: der *Checkin Wizard* ist für die Integration von Simulationsprogrammen zuständig, das *Model Setup Widget* dient hauptsächlich der Konfiguration der Simulation und das *Monitor Widget* gestattet eine Überwachung des Modellmanagements und der Simulation.

Hervorzuheben ist in diesem Zusammenhang, daß alle Objekte des Modellmanagements über die Benutzerschnittstelle zugänglich sind, so daß eine vollständige Einsicht und Transparenz für den Administrator oder einen versierten Nutzer ermöglicht wird. Außerdem wurden die unterschiedlichen Wissensstände des Menschen derart berücksichtigt, daß ein Benutzer nicht direkt Prozesse oder Prozeßstrukturen auswählen muß, sondern inhaltliche Aufgaben oder die Hardware im Cockpit, die im Zusammenhang mit der inhaltlichen Aufgabe steht. Die Assoziation der Aufgaben mit den entsprechenden Simulationsprozessen wird vom Modellmanagement geleistet (Prinzip der Selektoren).

Da ein Großteil der bearbeiteten Informationen dauerhaft zur Verfügung stehen muß, aber nicht vollständig zur Laufzeit neu gewonnen werden kann, ist *NeMO* an ein Datenbank-System angeschlossen worden. Damit existiert ein „Gedächtnis“, welches die Benutzung der Simulation durch Nicht-Experten ermöglicht sowie die Übergabe der Simulation an nachfolgende Mitarbeiter gewährleistet.

Der Natur der Daten entsprechend besteht *NeMO's Simulation Database* (NeSD) aus drei eigenständigen Teil-Datenbanken: *NeMO's Management Database* (NeMD), *NeMO's Program Database* (NePD) und *NeMO's Communication Database* (NeCD). Erstere enthält Information, die unmittelbar mit den Aufgaben und dem Aufbau des Modellmanagements in Beziehung stehen. Bei der Programm-Datenbank handelt es sich im Prinzip um ein System zur externen Dateiverwaltung der Simulationsprogramme, während mit Hilfe der Kommunikations-Datenbank alle in der Simulation ausgetauschten Daten in der Art eines *Interface Control Document's* (ICD) registriert werden.

Für die Zukunft steht primär die Portierung des Modellmanagements auf andere Plattformen an, da am Fachgebiet verstärkt Tendenzen zu erkennen sind, neben der ursprünglichen Entwicklungsplattform *IRIX* von *Silicon Graphics* auch Rechner mit anderen Betriebssystemen in der Simulation einzusetzen.

Weiterhin kann der Grad der durch das Modellmanagement in die Simulation ein-

geführten Automation erhöht werden. Dazu zählt z.B. die Auswertung der Informationen über die Datenwege zwischen Prozessen zur Laufzeit, um ohne Eingriff des Menschen für eine optimale Verteilung der Prozesse im Rechnernetzwerk zu sorgen, was auch im Zusammenhang mit der Synchronisation der Prozesse durch das Datenmanagement von Bedeutung ist. Ein weiterer Aspekt in dieser Hinsicht ist die Berücksichtigung der Belastung der Rechner, infolge derer das Modellmanagement automatisch eine Migration von Prozessen zu anderen Rechnern durchführen könnte.

Schließlich könnte das in dieser Arbeit vorgestellte Rechnermodell derart erweitert werden, so daß über das Modellmanagement auch eine Konfiguration der Rechner durchgeführt werden kann. Momentan handelt es sich bei dem Modell um eine reine Wiedergabe der Konfiguration des Computers, infolgedessen *NeMO* an dieser Stelle nur als Beobachter agieren kann. Fehlerhafte Einstellungen der Rechner können nicht selbständig vom Modellmanagement behoben werden.

Literatur

[ACE] ACE Online Documentation. Department of Computer Science, Washington University of St. Louis, http://www.cs.wustl.edu/~schmidt/ACE.html.

[AH01] G. Aschemann und P. Hasselmeyer. A Loosely Coupled Federation of Distributed Managment Services. *Journal of Network and Systems Management (JNSM)*, 9(1), 51–65, March 2001.

[Alb01] O.F. Albert. *Entwicklung einer Versuchsumgebung zur Untersuchung neuartiger Mensch-Maschine-Schnittstellen für Verkehrsflugzeugcockpits.* Dissertation in Vorbereitung, TU-Darmstadt, Fachbereich Maschinenbau, voraus. Erscheinungstermin 2001.

[AMV96] R. Aversa, N. Mazocca, und U. Villano. Design of a simulator of heterogenous computing environments. *Simulation Practice and Theory*, 4(2-3), 97–117, May 1996.

[And00] R. Anderl. Skriptum zur Vorlesung Produktdatentechnologie C. Technische Universität Darmstadt, Fachbereich Maschinenbau, Fachgebiet Datenverarbeitung in der Konstruktion, 2000.

[ARE+98] O.F. Albert, F. Roshani, K. Engels, C. Huth, F. Thurecht, und J. Schiefele. Das Forschungscockpit der TU Darmstadt, ein Werkzeug zur Untersuchung neuer Cockpitkonzepte. *DGLR-Fachausschußsitzung Luft- und Raumfahrt, Stuttgart*, Oktober 1998.

[AW98] J. Aronson und D. Wade. Model Based Simulation Composition. *Fall Simulation Interoperability Workshop*, 1998.

[BAA+92] Ö. Babaoglu, L. Alvisi, A. Amoroso, R. Davoli, und L.A. Giachini. Paralex: An Environment for Parallel Programming in Distributed Systems. In *Proceedings of the 6th ACM International Conference on Supercomputing*, Seiten 178–187. IEEE, July 1992.

[BBD99] R.E. Barker, P.T. Barham, und J.R. Damiano. The Dynamic Simulation Environment (DSE) Flexible Federation Object Model (FOM) Interface. *Spring Simulation Interoperability Workshop*, 1999.

[BBER+97] M.A. Bauer, R.B. Bunt, A. El Rayess, P.J. Finnigan, T. Kunz, H.L. Lutfiyya, A.D. Marshall, P. Martin, G.M. Oster, W. Powley, J. Rolia, D. Taylor, und M. Woodside. Services supporting management of distributed applications and systems. *IBM Systems Journal*, 36(4), June 1997.

[BEDM98] P. Benjamin, M. Erraguntla, D. Delen, und R. Mayer. Simulation Modeling at Multiple Levels of Abstraction. *Proceedings Winter Simulation Conference*, 1998.

[BGK+97] D. Bäumer, G. Gryczan, R. Knoll, C. Lilienthal, D. Riehle, und H. Züllighoven. Framework Development for Large Systems. *Communications of the ACM*, 40(10), Oktober 1997.

[BGL+99] W.G. Bleek, G. Gryzcan, C. Lilienthal, M. Lippert, S. Roock, H. Wolf, und H. Züllighoven. Frameworkbasierte Anwendungsentwicklung (Teil 2): Die Konstruktion interaktiver Anwendungen. *OBJEKTspektrum*, 2, 78–83, 1999.

[BHZ93] R. Biddle, J. Hine, und Z. Zhang. CR: A Monitor for Distributed Systems. In *Proceedings of the 10th Uniforum NZ Conference*, 1993.

[BK00] M. Brasse und N. Kuijpers. Realizing a Platform for Collaborative Virtual Environments based on High Level Architecture. *Spring Simulation Interoperability Workshop*, 2000.

[BLR+99] W.G. Bleek, M. Lippert, S. Roock, W. Strunk, und H. Züllighoven. Frameworkbasierte Anwendungsentwicklung (Teil 3): Die Anbindung von Benutzungsoberflächen und Entwicklungsumgebungen an Frameworks. *OBJEKTspektrum*, 3, 90–95, 1999.

[BLZ99] W.G. Bleek, C. Lilienthal, und H. Züllighoven. Frameworkbasierte Anwendungsentwicklung (Teil 4): Fachwerte. *OBJEKTspektrum*, 5, 1999.

[BN98] R. Beraldi und L. Nigro. Performance of a Time Warp based simulator of large scale PCS networks. *Simulation Practice and Theory*, 6(2), 149–163, February 1998.

[Boy96] R. Boys. What to Look for in a Reconfigurable Simulator. *AAAA Symposium*, 1996.

[Bäu98] D. Bäumer. *Softwarearchitekturen für die rahmenwerksbasierte Konstruktion großer Anwendungssysteme.* Dissertation, Universität Hamburg, Fachbereich Informatik, 1998.

[Bul94] H.J. Bullinger. *Ergonomie.* B.G. Teubner, Stuttgart, 1994. ISBN: 3-519-06366-2.

[Cal94] J.R. Callahan. Software Packager User's Guide. Bericht, NASA/WVU Software Research Lab, Fairmont, West Virginia, 1994. Technical Report # NASA-IVV-94-006.

[CB93] T. Clark und K.P. Birman. Using the ISIS Resource Manager for Distributed Fault-Tolerant Computing. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, Seiten 257–265, Los Alamitos, California, January 1993. IEEE, IEEE Computer Society Press.

[CB96] M.J. Corbin und G.F. Butler. MulTiSIM: An Object-Based Distributed Framework for Mission Simulation. *Simulation Practice and Theory*, 3(6), 383–399, January 1996.

[CH99] M. Cusack und P. Hoare. Component based development in HLA using JAVA. *Spring Simulation Interoperability Workshop*, 1999.

[Cla99] L. Clark. Honeywell Primus Epic - A Human Centered Cockpit Design. *4. Workshop Cockpit*, Februar 1999. TU Darmstadt.

[COR] Formal CORBA Specification. Object Management Group (OMG), http://www.omg.org/technology/documents/formal/.

[Cox98] K. Cox. A Framework-based Approach to HLA Federate Development. *Fall Simulation Interoperability Workshop*, 1998.

[CP94] J.R. Callahan und J.M. Purtillo. Using an architectural approach to integrate heteregeneous, distributed software components. Bericht, NASA/WVU Software Research Lab, Fairmont, West Virginia, 1994. Technical Report # NASA-IVV-94-003.

[CP98] A. Cox und M.D. Petty. Use of DIS Simulation Management (SIMAN) in HLA Federations. *1998 Spring Simulation Interoperability Workshop*, 1998.

[DA99] D. Davis und J. Aronson. Component Selection Techniques to Support Composable Simulation. *Spring Simulation Interoperability Workshop*, 1999.

[Dah98] J. Dahmann. High Level Architecture. *I/ITSEC Tutorial*, November 1998.

[Dal99] W.K. Dalheimer. *Programming with Qt*. O´Reilly, April 1999. ISBN: 1-56592-588-2.

[DMS] High Level Architecture - Technical Specifications. Defense Modeling And Simulation Office (DMSO), http://hla.dmso.mil.

[Eng01] K. Engels. *Realisierung und Untersuchung der Kommunikationsstruktur einer Simulationsarchitektur für einen verteilten Forschungssimulator*. Dissertation, TU-Darmstadt, Fachbereich Maschinenbau, 2001.

[FDG93] M. Flatebo, A.K. Datta, und S. Ghosh. Self-Stabilisation in Distributed Systems. In T.L. Casavant und M. Singhal, (Hrsg.), *Readings in Distributed Computing Systems*, Seiten 100–114. IEEE Computer Society Press, 1993.

[FR95] R.J. Friedrich und J.A. Rolia. Performance Evaluation of a Distributed Application Performance Monitor. Bericht, Hewlett Packard, December 1995.

[Gas97] T. Gassert. *Berechnung der Kurvenflugleistung von Segelflugzeugen*. Dissertation, TU-Darmstadt, Fachbereich Maschinenbau, 1997.

[GGBR98] S.M. Goss, P.L. Gustavson, J.T. Bachman, und L.M. Root. HLA Computer Aided Federated Development Environment (CAFDE). *Spring Simulation Interoperability Workshop*, 1998.

[GGGH98] H.M. Gullette, E. Goit, B. Gajkowski, und G. Hoyt. Distributed Simulation Exercise Construction Toolset (DiSECT). *Fall Simulation Interoperability Workshop*, 1998.

[GHR+00] G. Gryzcan, A. Havenstein, S. Roock, I. Wetzel, und H. Züllighoven. Frameworkbasierte Anwendungsentwicklung (Teil 5): Unterstützung von Kooperation mit persistenten fachlichen Behältern. *OBJEKTspektrum*, 1, 2000.

[GLL+99] G. Gryzcan, C. Lilienthal, M. Lippert, S. Roock, H. Wolf, und H. Züllighoven. Frameworkbasierte Anwendungsentwicklung (Teil 1). *OBJEKTspektrum*, 1, 1999.

[Gri99] F. Griffel. Komponenten - Softwarebausteine des nächsten Jahrtausends? *OBJEKTspektrum*, 1, 1999.

[GRWZ00] G. Gryzcan, S. Roock, H. Wolf, und H. Züllighoven. Frameworkbasierte Anwendungsentwicklung (Teil 6): Frameworkentwicklung und -einsatz organisieren. *OBJEKTspektrum*, 2, 2000.

[Ham97] M. Hammer. *Stereoskopische Informationsdarstellung am Beispiel eines Flugführungsdisplays.* Dissertation, TU-Darmstadt, Fachbereich Maschinenbau, 1997.

[HC01] G.T. Heinemnan und W.T. Councill. *Component-Based Software Engineering: Putting the Pieces Together.* Addison-Wesley, 2001. Kapitel 31: Overview of the CORBA Component Model.

[Hei99] V. Heil. Aktuelle Beispiele zur kooperativen Nutzung von Cockpit- und Bodensystemen zur Steigerung der Kapazität und Ökonomie ohne Beeinträchtigung der Sicherheit. *4. Workshop Cockpit*, Februar 1999. TU Darmstadt.

[HLSK98] S. Haines, T. Longshaw, R. Sleep, und R. Kennaway. An Open Architecture Approach to AAR. *1998 Spring Simulation Interoperability Workshop*, 1998.

[Hof00] R. Hoffman. Unterlagen zur Vorlesung Rechnerarchitektur. Technische Universität Darmstadt, Fachbereich Informatik, Fachgebiet Rechnerarchitektur, 2000.

[Hua97] J.Y. Huang. Design of a Plug and Play Simulator over Distributed Environment. *Fall Simulation Interoperability Workshop*, 1997.

[JP99] L.A. Jackson und G.M. Pearman. A Low Level Distributed Simulation Architecture. *Fall Simulation Interoperability Workshop*, 1999.

[Kau98] R. Kaufhold. *Konzeption und Erprobung der Visualisierung von Geländeinformationen in Flugführungsanzeigen.* Dissertation, TU-Darmstadt, Fachbereich Maschinenbau, 1998. Shaker Verlag.

[Kin98] J.A. King. Use of Distributed Architecture to implement a Reconfigurable Simulator. *1998 IMAGE Conference*, August 1998.

[KMSD93] J. Kramer, J. Magee, M. Sloman, und N. Dulay. Configuring Object-Based Distributed Programs in REX. In T.L. Casavant und M. Singhal, (Hrsg.), *Readings in Distributed Computing Systems*, Seiten 187–205. IEEE Computer Society Press, 1993.

[KPSH95] J. Knappen, H. Partl, E. Schlegl, und I. Hyna. LaTeX2e - Kurzbeschreibung. Bericht, 1995.

[KSKP97] K.H. Kim, Y.R. Seong, T.G. Kim, und K.H. Park. Ordering of simultaneous events in distributed DEVS simulation. *Simulation Practice and Theory*, 5(3), 253–268, March 1997.

[Kub96] W. Kubbat. Skriptum zur Vorlesung Prozeßdatenverarbeitung I+II. Technische Universität Darmstadt, Fachbereich Maschinenbau, Fachgebiet Flugmechanik und Regelungstechnik, 1996.

[Kub98] W. Kubbat. Skriptum zur Vorlesung Flugmechanik I+II. Technische Universität Darmstadt, Fachbereich Maschinenbau, Fachgebiet Flugmechanik und Regelungstechnik, 1998.

[Kub99] W. Kubbat. Ikarus, wohin? Zur Situation der Luftfahrtforschung an deutschen Hochschulen. *Jahrestagung des Luftfahrt-Presseclubs, Stuttgart*, 1999.

[Kun93] T. Kunz. Abstract Debugging of Distributed Applications. *Institut für Theoretische Informatik, Fachbereich Informatik, Technische Hochschule Darmstadt*, Dezember 1993.

[KvGJ98] N. Kuijpers, P. van Gool, und H. Jense. A Component Architecture for Simulator Development. *Spring Simulation Interoperability Workshop*, 1998.

[LKG91] F. Lange, R. Kroeger, und M. Gergeleit. JEWEL: Design and Implementation of a Distributed Measurment System. *GMD*, 1991.

[Lou98] D. Louis. *C und C++: Programmierung und Referenz.* Markt und Technik, Buch- und Software Verlag, 1998.

[Lut97] R. Lutz. A Comparison Of HLA Object Modeling Principles With Traditional Object-Oriented Modeling Concepts. *Fall Simulation Interoperability Workshop*, 1997.

[Mar01] M.P. Marpert. Entwicklung einer Signaturbeschreibung für Simulationsprozesse in der verteilten Simulationsarchitektur DSPA. Studienarbeit, Technische Universität Darmstadt, Fachgebiet Flugmechanik und Regelungstechnik, 2001.

[MB98] S.R. Moore und J.E. Bell. A Simulation Execution Environment for STOW97. *Spring Simulation Interoperability Workshop*, 1998.

[MCWB93] K. Marzullo, R. Cooper, M.D. Wood, und K.P. Birman. Tools for Distributed Application Management. In T.L. Casavant und M. Singhal, (Hrsg.), *Readings in Distributed Computing Systems*, Seiten 311–326. IEEE Computer Society Press, 1993.

[MHGM00] B. McCauley, J. Hill, P. Gravitz, und D. McFarland. JMASS98 - Engagement Level Simulation Framwork++. *Spring Simulation Interoperability Workshop*, 2000.

[Mic99] J. Mickel. Zukünftige Aufgaben und Tätigkeiten der airborne crew am Beispiel des NEAP und FREER II Projekts. *4. Workshop Cockpit*, Februar 1999. TU Darmstadt.

[MMRV97] A. Mazzeo, N. Mazzocca, S. Russo, und V. Vittorini. A method for predictive performance of distributed programs. *Simulation Practice and Theory*, 5(1), 65–82, January 1997.

[MPBR93] B.E. Martin, C.H. Pedersen, und J. Bedford-Roberts. An Object-Based Taxonomy for Distributed Computing Systems. In T.L. Casavant und M. Singhal, (Hrsg.), *Readings in Distributed Computing Systems*, Seiten 152–169. IEEE Computer Society Press, 1993.

[Mul97] P.A. Muller. *Instant-UML*. Wrox Press, 1997.

[Nau99] J.M. Naud. Simulation Models as Components in an HLA World. *Spring Simulation Interoperability Workshop*, 1999.

[NIS] Dictionairy of Algorithms, Data Structures and Problems. NIST - National Institute of Standards and Technology, http://hissa.nist.gov/dads.

[NO95] R.E. Nance und C.M. Overstreet. Computer Simulation: Achieving Credible Experimental Results in Virtual Environments. June 1995. http://www.src.vt.edu/SRC_Technical_Reports.html.

[OTW] OTW - Objekttechnologie Werkbank 2.4 Benutzerhandbuch. UML Modellierungswerkzeug, OWiS Software Gmbh, http://www.otwsoftware.com.

[Pac97] D.K. Pace. Fidelity Considerations for RDE Distributed Simulation. *97 Fall Simulation Interoperabilty Workshop Papers*, 1, 249–259, September 1997.

[Pac98] D.K. Pace. Dimensions and Attributes of Simulation Fidelity. *1998 Fall Simulation Interoperability Workshop*, 1998.

[PLM88] D. Parkinson und S. Link-Miles. Functionally Distributed Simulation. *Flight Simulation: Recent Developments in Technology and Use: Two Day Conference*, Seiten 3–7, 1988.

[Pur00] M. Purpus. *Die Rolle der Prädiktion in dreidimensionalen Flugführungsdarstellungen.* Dissertation, TU-Darmstadt, Fachbereich Maschinenbau, 2000.

[Qt00] Qt Online Reference Documentation. Trolltech AS, Computer Software Company, http://doc.trolltech.com, 2000.

[QtD01] Online Manual für Qt GUI Designer. Trolltech AS, Computer Software Company, http://doc.trolltech.com/designer.html, 2001.

[Raj98a] G.S. Raj. A Detailed Comparison of CORBA, DCOM and JAVA/RMI. *Web Cornucopia*, 1998. http://www.execpc.com/˜gopalan/misc/compare.html.

[Raj98b] G.S. Raj. A Detailed Comparison of Enterprise JavaBeans (EJB) and The Microsoft Transaction Server (MTS) Models. *Web Cornucopia*, 1998.

[SBB93] M.K. Saxena, K.K. Biswas, und P.C.P. Bhatt. Problem Solving with Distributed Knowledge. In T.L. Casavant und M. Singhal, (Hrsg.), *Readings in Distributed Computing Systems*, Seiten 267–284. IEEE Computer Society Press, 1993.

[Sch] D.C. Schmidt. Developing Distributed Object Computing Applications with CORBA. *Online Tutorial*. Washington University, St. Louis, http://www.cs.wustl.edu/~schmidt/PDF/corba4.pdf.

[Sch85] B. Schmidt. *Systemanalyse und Modellaufbau: Grundlagen der Simulationstechnik*. Springer-Verlag, 1985.

[Sch00a] J. Schiefele. *Realization and Evaluation of Virtual Cockpit Simulation and Virtual Flight Simulation*. Dissertation, TU-Darmstadt, Fachbereich Maschinenbau, 2000. Shaker Verlag.

[Sch00b] J. Schimmel. CORBA Component Model. Seminarausarbeitung - Kommunikation in verteilten Systemen, Informationstechnologie Transfer Office, TU Darmstadt, Juli 2000.

[Sie96] J. Siegel. *CORBA: Fundamentals and Programming*. John Wiley & Sons, 1996.

[SK99] S. Straßburger und U. Klein. Introduction to the High Level Architecture for Modeling and Simulation (HLA). *1999 European Simulation Multiconference, Warschau, Polen*, June 1999.

[Smi99] R. Smith. Simulation Article. *Encyclopedia in Computer Science, Macmillan Reference Press, New York*, November 1999.

[Sta00] M. Stal. Reich der Mitte: Die Komponententechnologien COM+, EJB und CORBA Components. *OBJEKTspektrum*, 3, 2000.

[STL] Standard Template Library (STL) Programmer's Guide. Silicon Graphics, Inc., http://www.sgi.com/tech/stl.

[Syy98] U. Syyid. The Adaptive Communication Environment: ACE, A Tutorial. Bericht, Hughes Network Systems, http://www.cs.wustl.edu/~schmidt/PDF/ACE-tutorial.pdf, 1998.

[Tan95] A.S. Tanenbaum. *Verteilte Betriebssysteme*. Prentice-Hall, 1995.

[TAO] TAO Online Documentation. Department of Computer Science, Washington University of St. Louis, http://www.cs.wustl.edu/~schmidt/TAO.html.

[WH98] D. Wright und C. Harris. Determining and Expressing RTI Requirements. *1998 Spring Simulation Interoperability Workshop*, 1998.

[Wil93] C.E. Wills. A Model for Executing Computations in a Distributed Environment. In T.L. Casavant und M. Singhal, (Hrsg.), *Readings in Distributed Computing Systems*, Seiten 116–132. IEEE Computer Society Press, 1993.

[WLS00] N. Wang, D. Levine, und D.C. Schmidt. Optimizing the CORBA Component Model for High-performance and Real-time Applications. *Middleware 2000 Conference*, April 2000.

[WM93] M. Wood und K. Marzullo. The Design and Implementation of Meta. In K.P. Birman und R. van Renesse, (Hrsg.), *Reliable Distributed Computing with the ISIS toolkit*, Seiten 309–327. IEEE Computer Society Press, 1993.

[WM98] E. White und M. Myiak. A Conceptual Model for Simulation Load Balancing. In *1998 Spring Simulation Interoperability Workshop*, 1998.

Index